The Big Book of PowerShell Gotchas

>

PowerShell.org

Table of Contents

PowerShell is full of "gotchas" - little things that just get in your way and are hard to figure out on your own. This short book is intended to help you figure them out and avoid them.

The Big Book of PowerShell Gotchas

by (mostly) Don Jones

PowerShell is full of "gotchas" - little things that just get in your way and are hard to figure out on your own. This short book is intended to help you figure them out and avoid them.

Getting the Code The EnhancedHTML2 module mentioned in this book can be found in the https://www.powershellgallery.com/packages/EnhancedHTML2/. That page includes download instructions. PowerShellGet is requires, and can be obtained from PowerShellGallery.com

Was this book helpful? The author(s) kindly ask(s) that you make a tax-deductible (in the US; check your laws if you live elsewhere) donation of any amount to The DevOps Collective to support their ongoing work.

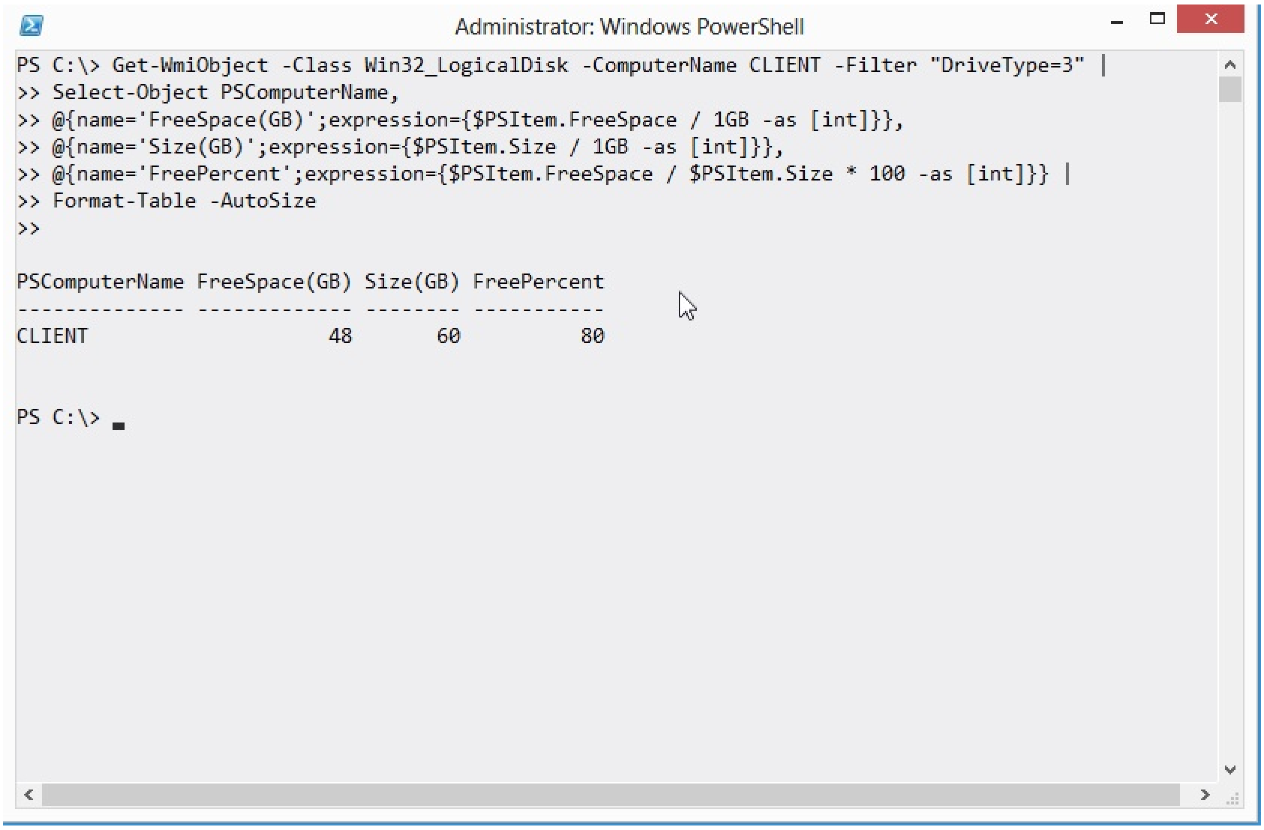
Check for Updates! Our ebooks are often updated with new and corrected content. We make them available in three ways:

- Our main, authoritative GitHub organization, with a repo for each book. Visit https://github.com/devops-collective-inc/
- Our GitBook page, where you can browse books online, or download as PDF, EPUB, or MOBI. Using the online reader, you can link to specific chapters. Visit https://www.gitbook.com/@devopscollective
- On LeanPub, where you can download as PDF, EPUB, or MOBI (login required), and "purchase" the books to make a donation to DevOps Collective. You can also choose to be notified of updates. Visit https://leanpub.com/u/devopscollective

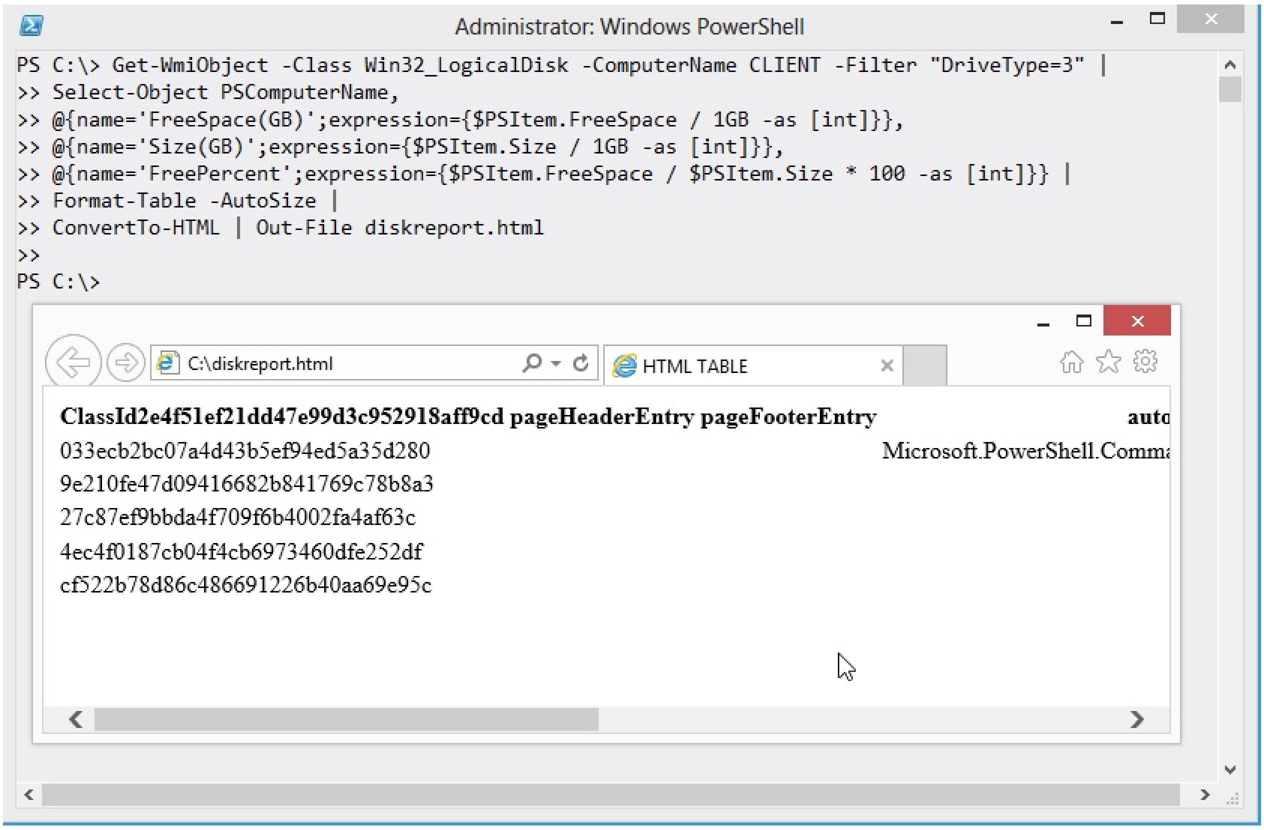
GitBook and LeanPub have slightly different PDF formatting output, so you can choose the one you prefer. LeanPub can also notify you when we push updates. Our main GitHub repo is authoritative; repositories on other sites are usually just mirrors used for the publishing process. GitBook will usually contain our latest version, including not-yet-finished bits; LeanPub always contains the most recent "public release" of any book.

Format right

Everyone runs into this one. Here's how it goes: you start by writing a truly awesome command.



And you think, "wow, that'd go great in an HTML file."



Wait... what?!?!?

This happens all the time. If you want an easy way to remember what not to do, it's this: Never pipe a Format command to anything else. That isn't the whole truth, and we'll get to the whole truth in a sec, but if you just want a quick answer, that's it. In the community, we call it the "Format Right" rule, because you have to move your Format command to the right-most end of the command line. That is, the Format command comes last, and nothing else comes after it.

The reason is that the Format commands all produce special internal formatting codes, that are really just intended to create an on-screen display. Piping those codes to anything else - ConvertTo-HTML, Export-CSV, whatever - just gets you gibberish output.

In fact, there are actually a few commands that can come after a Format command in the pipeline:

1. Out-Default. This is technically always at the end of the pipeline, although it's invisible. It redirects to Out-Host.
2. Out-Host also understands the output of Format commands, because Out-Host is how those formatting codes get on the screen in the first place.
3. Out-Printer understands the formatting codes too, and constructs a printed page that would look exactly like the normal on-screen output.
4. Out-File, like Out-Printer, redirects the on-screen output, but this time to a text file on disk.
5. Out-String consumes the formatting codes and just outputs a plain string containing the

text that would otherwise have appeared on-screen.

Apart from those exceptions - and of them, you'll mainly only ever use Out-File - you can't pipe the output of a Format command to much else and get anything that looks useful.

Where is the __ Command? I've Installed the Latest Version of PowerShell and Can't Find it!

One tricky thing is understanding that there are a certain number of commands that *come with PowerShell, _while _most* commands do not.

Every new version of PowerShell includes at least a few new commands. For example, Start-Job appeared for the first time in PowerShell v2, while Invoke-AsWorkflow was introduced in PowerShell v3.

What confuses people is that a new version of PowerShell also tends to correspond with a new version of the Windows operating system? and the OS itself comes with hundreds of commands. For example, you may have used Get-SmbShare for the first time in Windows Server 2012, which included PowerShell v3. But Get-SmbShare is *part of the operating system, not part of PowerShell.* That is, you won't have Get-SmbShare on every system that has PowerShell v3 or later, because the command isn't a "feature of PowerShell," it's a "feature of Windows."

So? where do you get commands?

Usually, with whatever product those commands are a part of. Want the Exchange Server commands? Install the Exchange Server admin tools. Want the Windows Server 2012 commands? Install the Remote Server Administration Toolkit (RSAT), which contains the server admin tools.

PowerShell.exe isn’t PowerShell

It’s important to understand that Windows PowerShell is actually an untouchable, behind-the-scenes engine. You as a mere human being cannot easily interact directly with PowerShell.

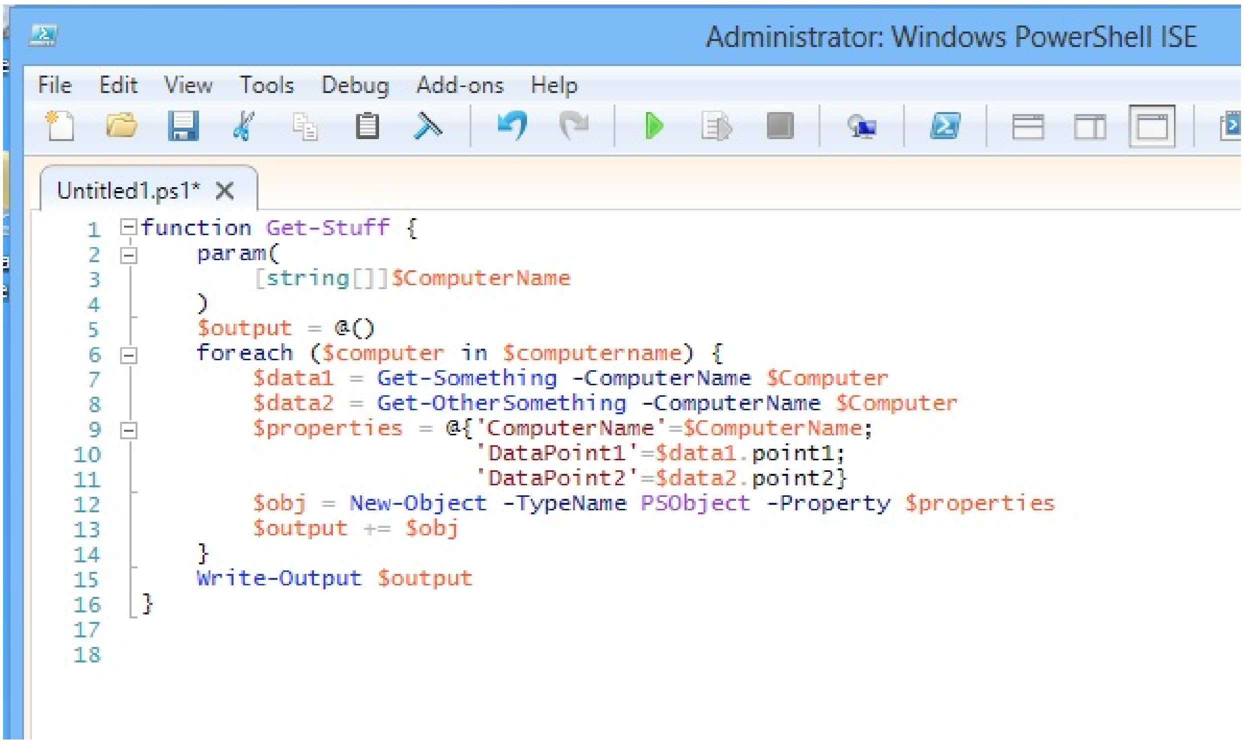
Instead, you need a host application. A host embeds the engine internally, and then gives you a way to interact with it. For example, PowerShell.exe is a host application. It is built around the same Windows console host (ConHost.exe) as the old Cmd.exe command-line shell, but it embeds the PowerShell engine. You type commands, and the host hands those to the engine for execution. The host is also responsible for displaying any results ? in this case, on-screen.

Why is this distinction important?

Because different hosts can behave in different ways. For example, the PowerShell ISE behaves a bit differently than the console host, and both of them behave very differently from Active Directory Administration Center ? another PowerShell host.

Accumulating Output in a Function

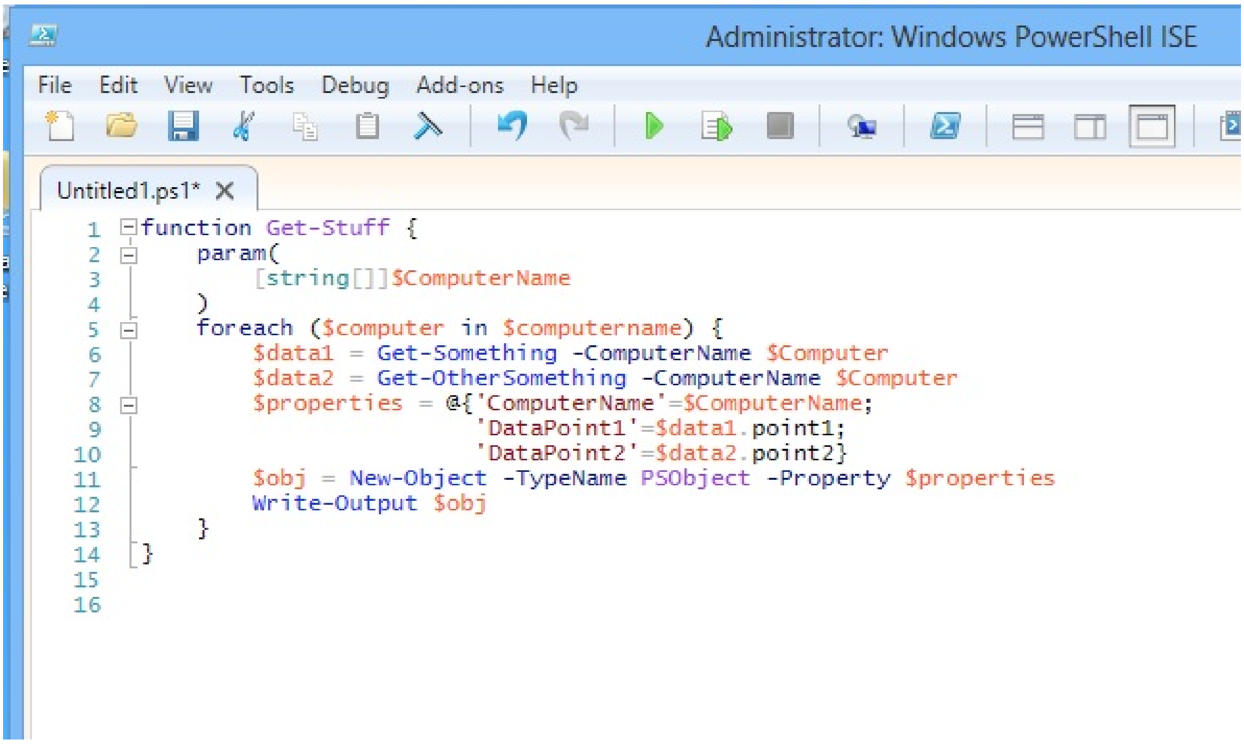
This is a bit of an "advanced" gotcha, but it's one that many experienced developers will run into. Here's a very trimmed-down example, just to make the point (it isn't functional, as the command used is fictional):



The problem here is that the function can generate multiple output objects, and the programmer is accumulating those into the $output variable. That means this function won't output anything until it's completely finished running. That isn't how PowerShell commands (and functions *are* commands) are usually meant to work.

PowerShell commands should *usually* output each object to the pipeline, one at a time, as those objects are ready. That allows the *pipeline* to accumulate the output, and to immediately pass it along to whatever is next in the pipeline. That's how PowerShell commands are intended to work. Now, there are always exceptions. Sort-Object, for example, *has* to accumulate its output, because it can't actually sort anything until it has *all* of them. So it's called a _blocking command, _because it "blocks" the pipeline from doing anything else until it produces its output. But that's an exception.

It's usually easy to fix this, by simply outputting to the pipeline instead of accumulating:

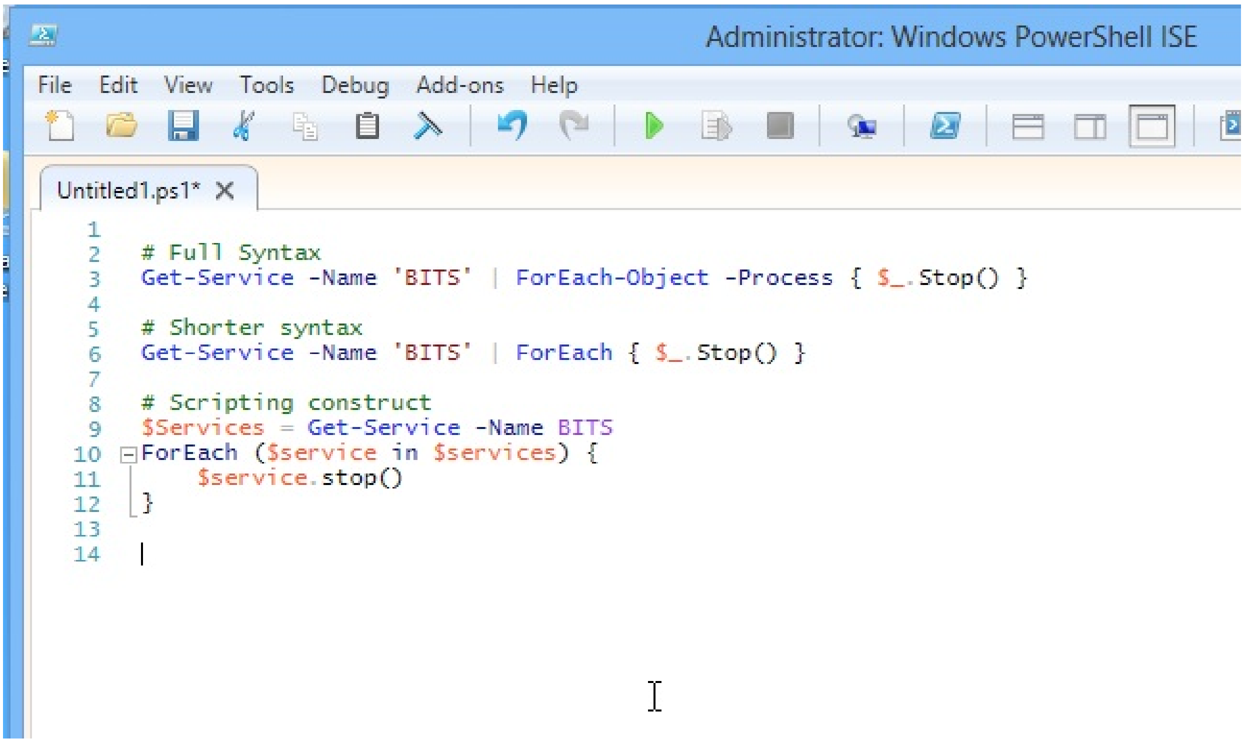

Administrator: Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
Untitled1.ps1*
function Get-Stuff {
param(
[string[]]$ComputerName
)
foreach ($computer in $computername) {
$data1 = Get-Something -ComputerName $Computer
$data2 = Get-OtherSomething -ComputerName $Computer
$properties = @{'ComputerName'=$ComputerName;
'DataPoint1'=$data1.point1;
'DataPoint2'=$data2.point2}
$obj = New-Object -TypeName PSObject -Property $properties
Write-Output $obj
}
}

ForEach vs ForEach vs ForEach

PowerShell's three lookalike friends can confusing, especially for newcomers. Basically, you've got two entities:

- The ForEach-Object cmdlet, which has an alias ForEach (it also has the alias %). This is meant to operate in the pipeline, and it uses a ?Process parameter that accepts a scriptblock.
- The ForEach scripting construct. This has a specific syntax, is not intended to be used in the pipeline, and does not have an alias.

Here's all three in action, in a very simplistic example:



The big difference is that, in the pipeline, ForEach-Object _processes one object at a time. _That means it can be slower, since that scriptblock must be interpreted on each iteration. It also tends to use less memory, since objects streaming down the pipeline one at a time don't all have to be bunched up in a variable first.

The scripting construct tends to be faster, but it often has more memory overhead, because you have to give it the entire collection of objects at once, instead of streaming objects into it one at a time.

Both use vaguely similar syntax, but there are differences. It's important to understand **that they are not the same,** and that they execute differently. It's confusing because "ForEach" is both an alias and a scripting construct; the shell determines which you're using by looking

at the context in which you're using it.

Tab Completion

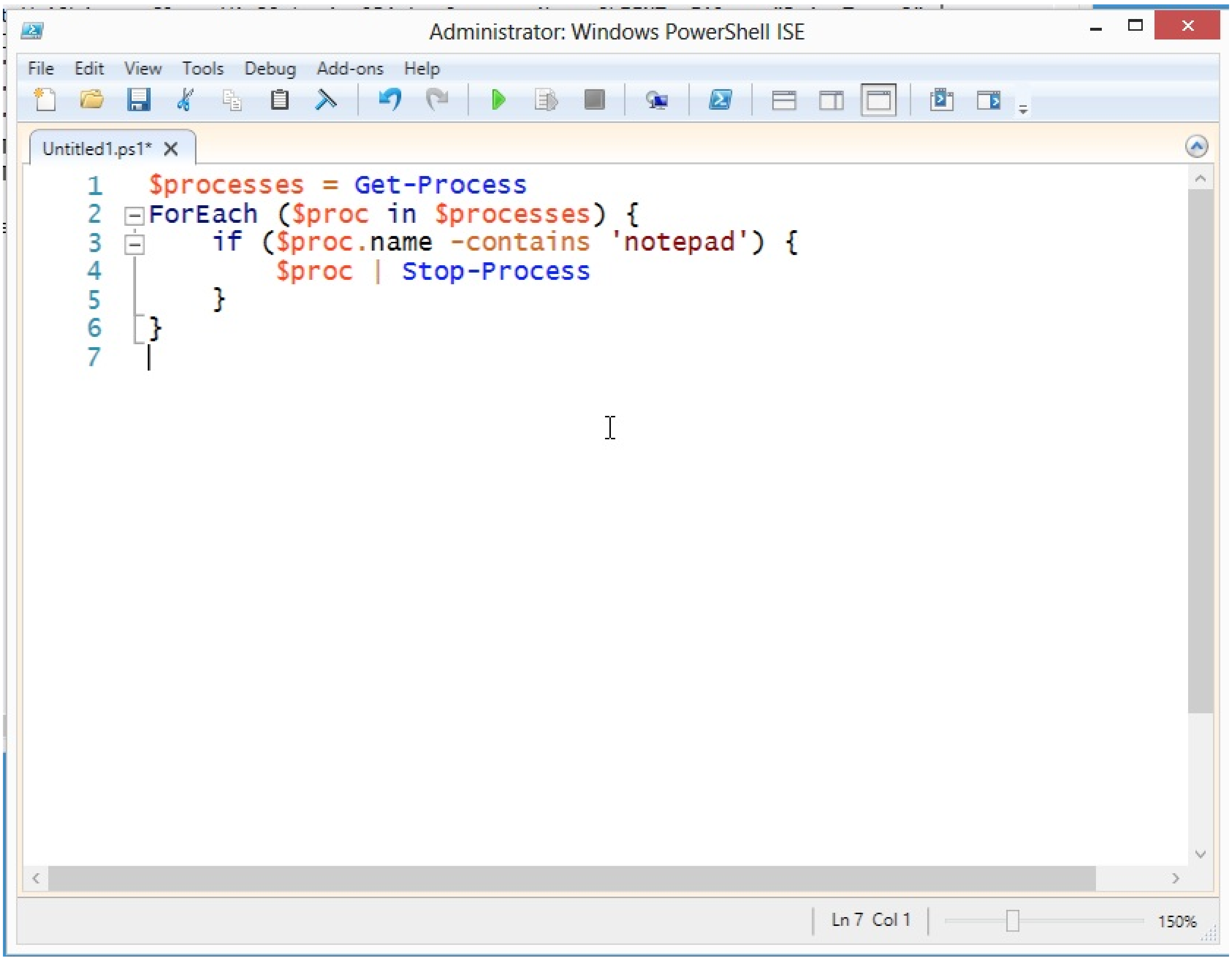
It's sad and amazing how few people rely on tab completion, both in the PowerShell ISE and in the console window.

- When you tab complete, you'll never spell commands or parameter names wrong
- For many parameter values that are static lists, or easily-queried lists, tab completion (especially in v3 and later) can fill-in legal parameter values for you
- Tab completion makes long cmdlet names a lot easier to type, without the need for difficult-to-remember aliases.

Get into the habit of using tab completion all the time, and you're guaranteed to make fewer mistakes.

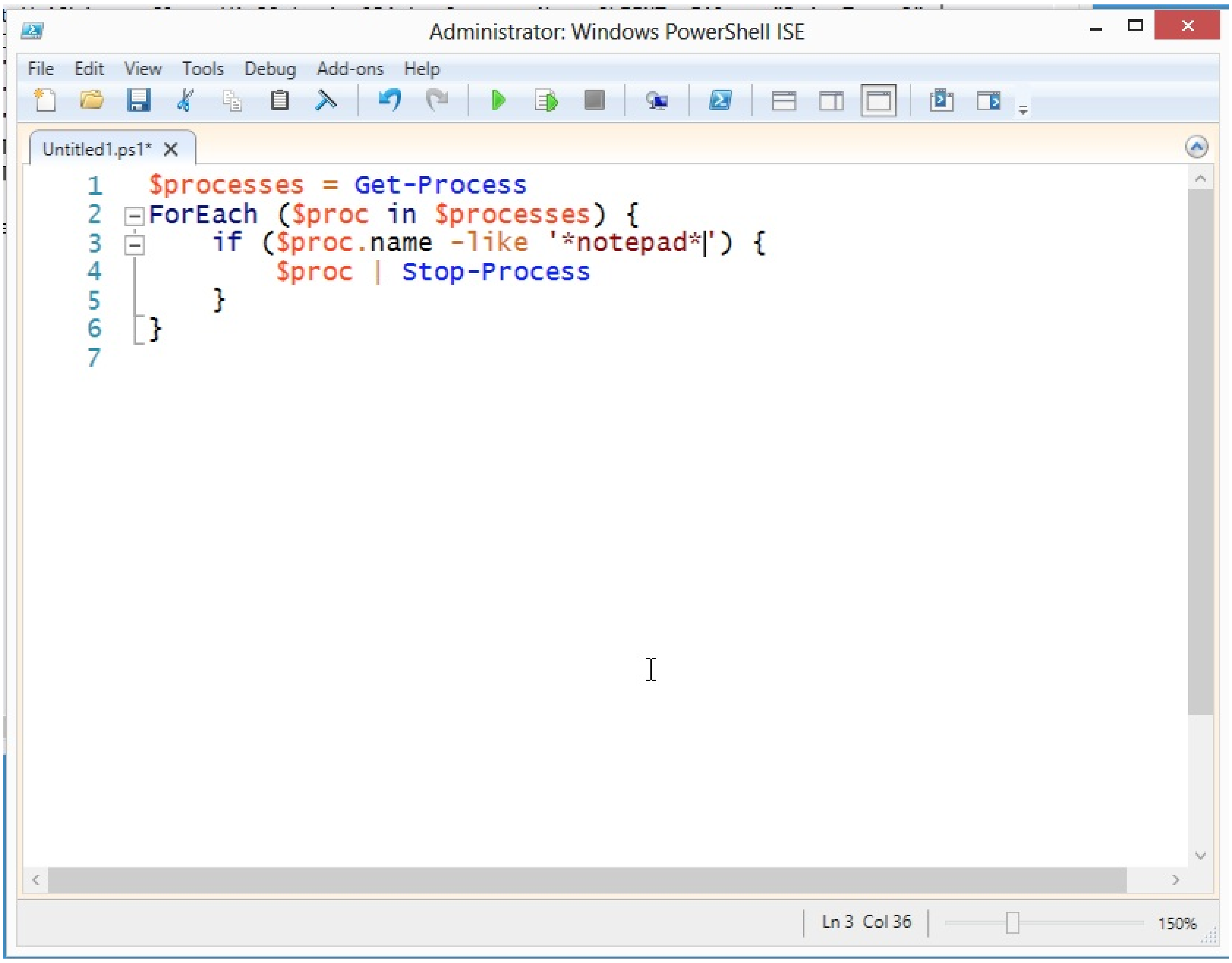
-Contains isn’t -Like

Oh, if I had a nickel for every time I've seen this:



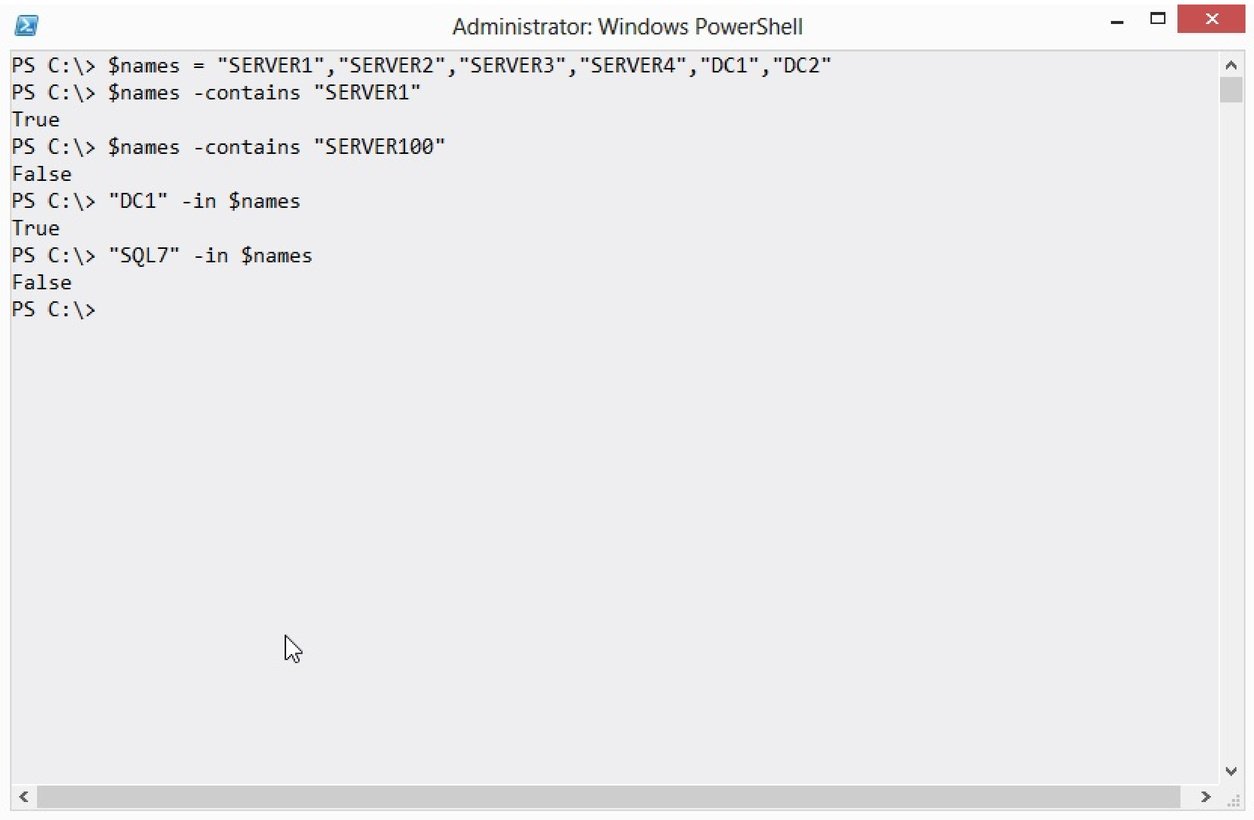
I get how this happens. The -contains operator seems like it should be checking to see if a process' name contains the letters "notepad." But that isn't what it does.

The correct approach is to use the -like operator, which in fact does do a wildcard string comparison:

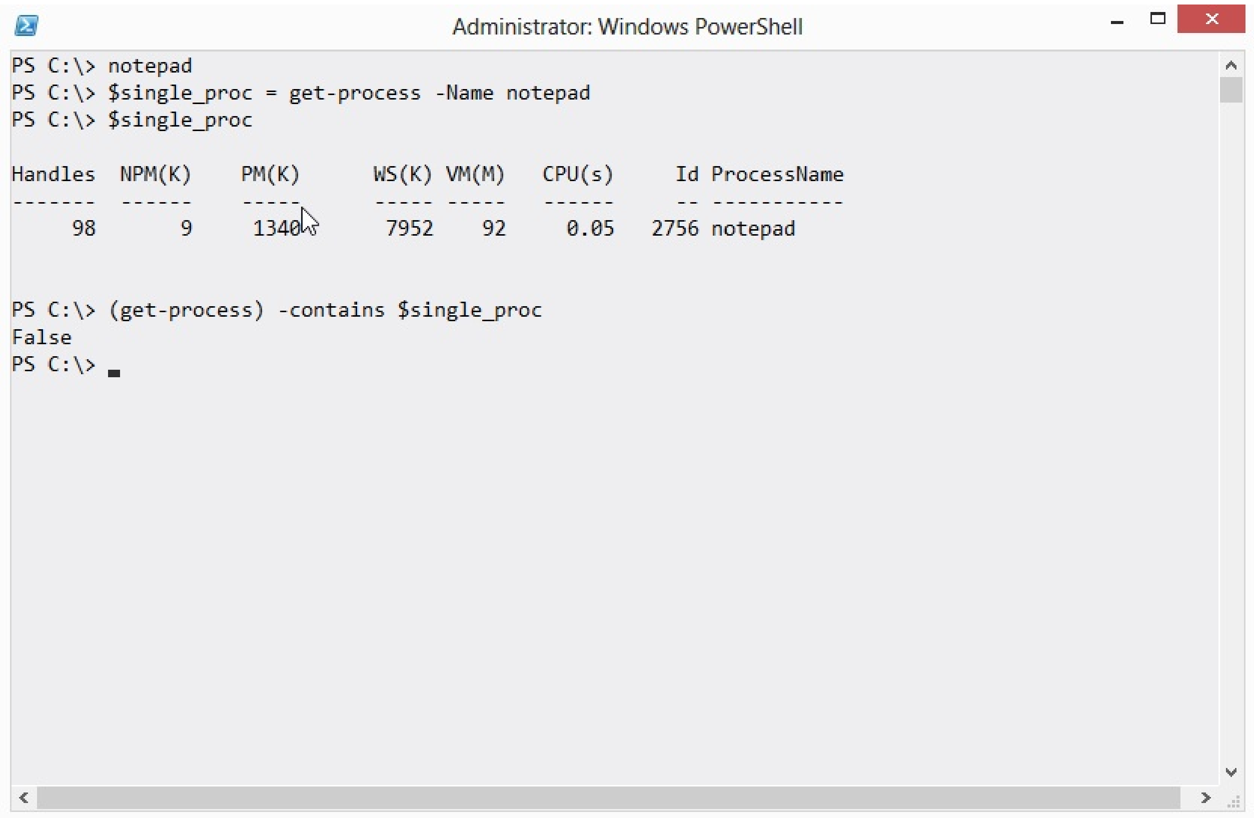


I'll let pass the thought that the really correct answer is to just run Stop-Process -name *notepad*, because I was aiming for a simple example here. But... don't overthink things. Sometimes a script and a ForEach loop isn't the best approach.

So anyway, what does -contains (and its friend, -notcontains) actually do? They're similar to the -in and -notin operators introduced in PowerShell v3, and those operators cause more than a bit of confusion, too. What they do is check to see if a collection of objects contains a given single object. For example:



In fact, that example is probably the best way to see it work. The trick is that, when you use a complex object instead of a simple value (as I did in that example), -contains and -in look at every property of the object to make a match. If you think about something like a process, they're always changing. From moment to moment, a process' CPU and memory, for example, are different.

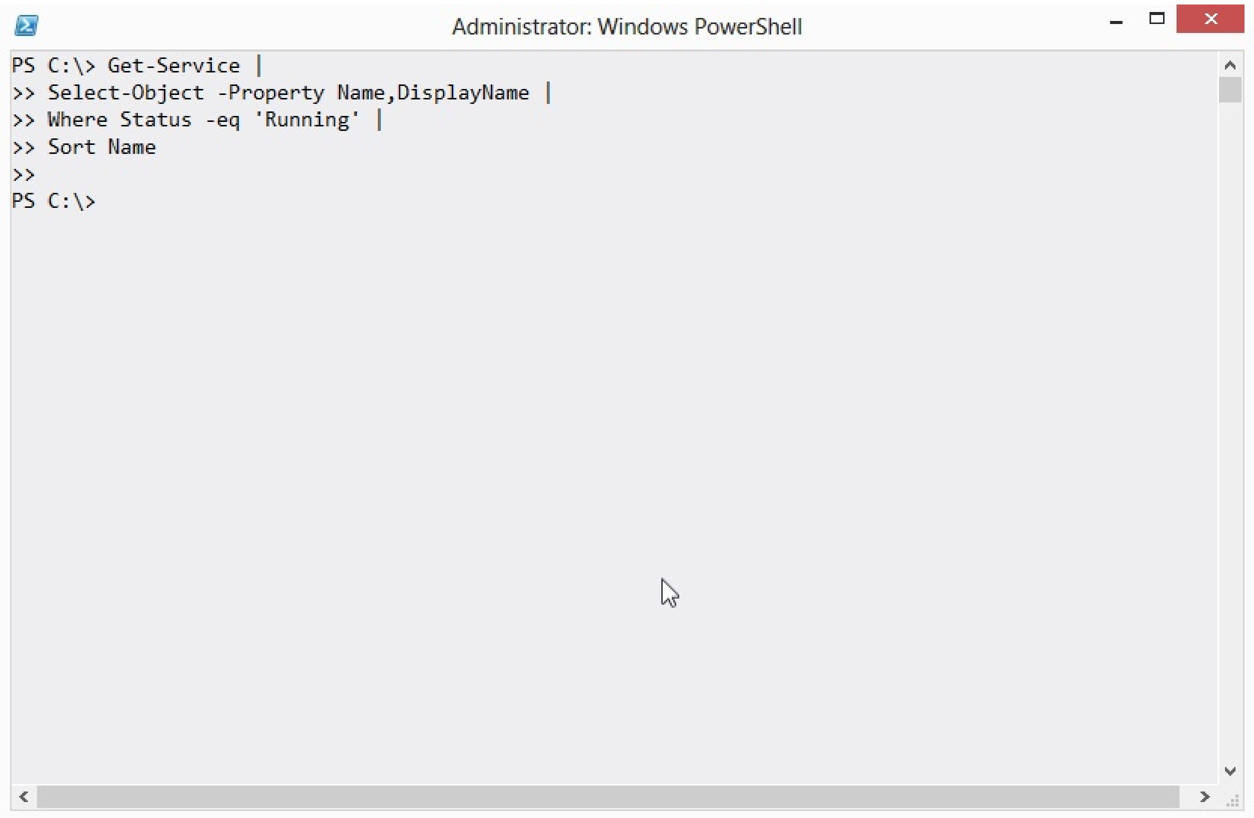


In this example, I've started Notepad. I've put its process object into $single_proc, and you can see that I verified it was there. But when I run Get-Process and check to see if its collection contained my Notepad, I got False. That's because the object in $single_proc is out of date. Notepad is running, but it now looks different, so -contains can't find the match.

The -in and -contains operators are best with simple values, or with objects that don't have constantly-changing property values. But they're not wild card string matching operators. Use -like (or -notlike) for that.

You Can’t Have What You Don’t Have

Can you see what's wrong with this approach?



I mean, I'm pretty sure I have some running services, which is what this was supposed to display.

If you don't see the answer right away - or frankly, even if you do - this is a good time to talk about how to troubleshoot long command lines. Start, as I always say, by backing off a step. Delete the last command, and see if that does anything different.

```
Administrator: Windows PowerShell
PS C:\> Get-Service |
>> Select-Object -Property Name,DisplayName |
>> Where Status -eq 'Running' |
>> Sort Name
>>
PS C:\> Get-Service |
>> Select-Object -Property Name,DisplayName |
>> Where Status -eq 'Running'
>>
PS C:\> Get-Service |
>> Select-Object -Property Name,DisplayName
>>

Name                                              DisplayName
----                                              -----------
AeLookupSvc                                       Application Experience
ALG                                               Application Layer Gateway Service
AllUserInstallAgent                               Windows All-User Install Agent
AppIDSvc                                          Application Identity
Appinfo                                           Application Information
AppMgmt                                           Application Management
AudioEndpointBuilder                              Windows Audio Endpoint Builder
Audiosrv                                          Windows Audio
AxInstSV                                          ActiveX Installer (AxInstSV)
BDESVC                                            BitLocker Drive Encryption Service
BFE                                               Base Filtering Engine
BITS                                              Background Intelligent Transfer Service
```

In this case, I removed the Sort-Object (Sort) command, and nothing different happened. So that wasn't causing the problem. Next, I removed the Where-Object (Where, using v3 short syntax) command, and ah-ha! I got output. So something broke with Where-Object. Let's take what did work and pipe it to Get-Member, to see what's in the pipeline after Select-Object runs.

```
Administrator: Windows PowerShell
wscsvc                                            Security Center
WSearch                                           Windows Search
WSService                                         Windows Store Service (WSService)
wuauserv                                          Windows Update
wudfsvc                                           Windows Driver Foundation - User-mode Driver F...
WwanSvc                                           WWAN AutoConfig


PS C:\> Get-Service |
>> Select-Object -Property Name,DisplayName |
>> Get-Member
>>


   TypeName: Selected.System.ServiceProcess.ServiceController

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
DisplayName NoteProperty System.String DisplayName=Application Experience
Name        NoteProperty System.String Name=AeLookupSvc


PS C:\>
```

OK, I have an object that has a DisplayName property and a Name property.

And my Where-Object command was checking the Status property. Do you see a Status property? No, you do not. My error is that I removed the Status property when I didn't include it in the property list of Select-Object. So Where-Object had nothing to work with, so it returned nothing.

(Yeah, it'd be cooler if it threw an error - "Hey, you said to filter on the Status property, and there ain't one!" - but that isn't how it works.)

Moral of the story: Pay attention to what's in the pipeline. You can't work with something you don't have, and you might have taken it away yourself. You won't always get a helpful error message, so sometimes you'll need to dig in and figure it out another way - such as backing off a step.

-Filter Values Diversity

Here's one of the toughest things to get used to in PowerShell:

```
Administrator: Windows PowerShell
PS C:\> Get-ChildItem -Filter *.html


    Directory: C:\


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         4/26/2013  12:07 PM       2148 diskreport.html


PS C:\> Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3"


DeviceID     : C:
DriveType    : 3
ProviderName :
FreeSpace    : 51293458432
Size         : 64055406592
VolumeName   :



PS C:\> Get-ADUser -Filter { title -eq 'CTO' }
```

Here you see three commands, each using a -Filter parameter. Every one of those filters is different.

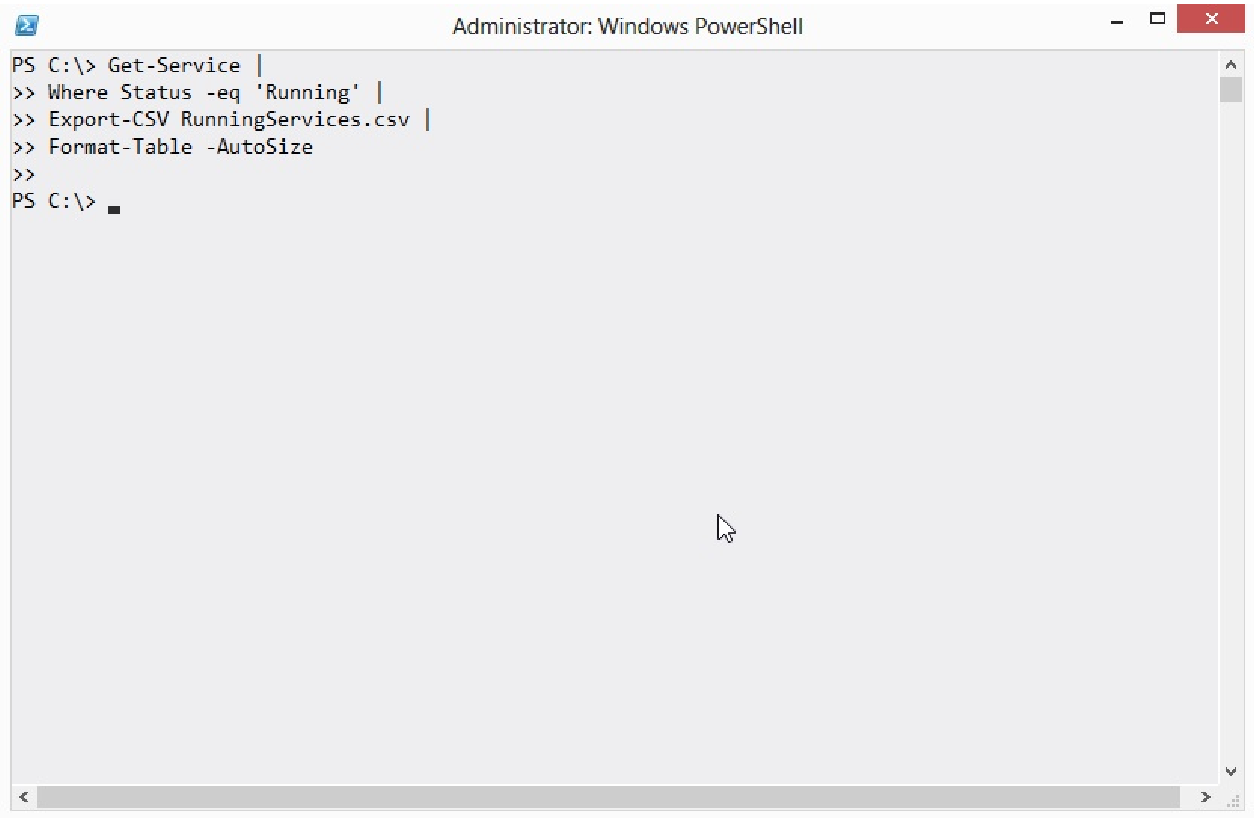
1. With Get-ChildItem, -Filter accepts file system wildcards like *.
2. With Get-WmiObject, -Filter requires a string, and uses programming-style operators (like = for equality).
3. With Get-ADUser, -Filter wanted a script block, and accepted PowerShell-style comparison operators (like -eq for equality).

Here's how I think of it: When you use a -Filter parameter, PowerShell isn't processing the filtering. Instead, the filtration criteria is being handed down to the underlying technology, like the file system, or WMI, or Active Directory. That technology gets to decide what kind of filter criteria it will accept. PowerShell is just the middleman. So you have to carefully read the help, and maybe look for examples, to understand how the underlying technology needs you to specify its filter.

Yeah, it'd be nice if PowerShell just translated for you (that's actually what Get-ADUser does - the command translates that into an LDAP filter under the hood). But, usually, it doesn't.

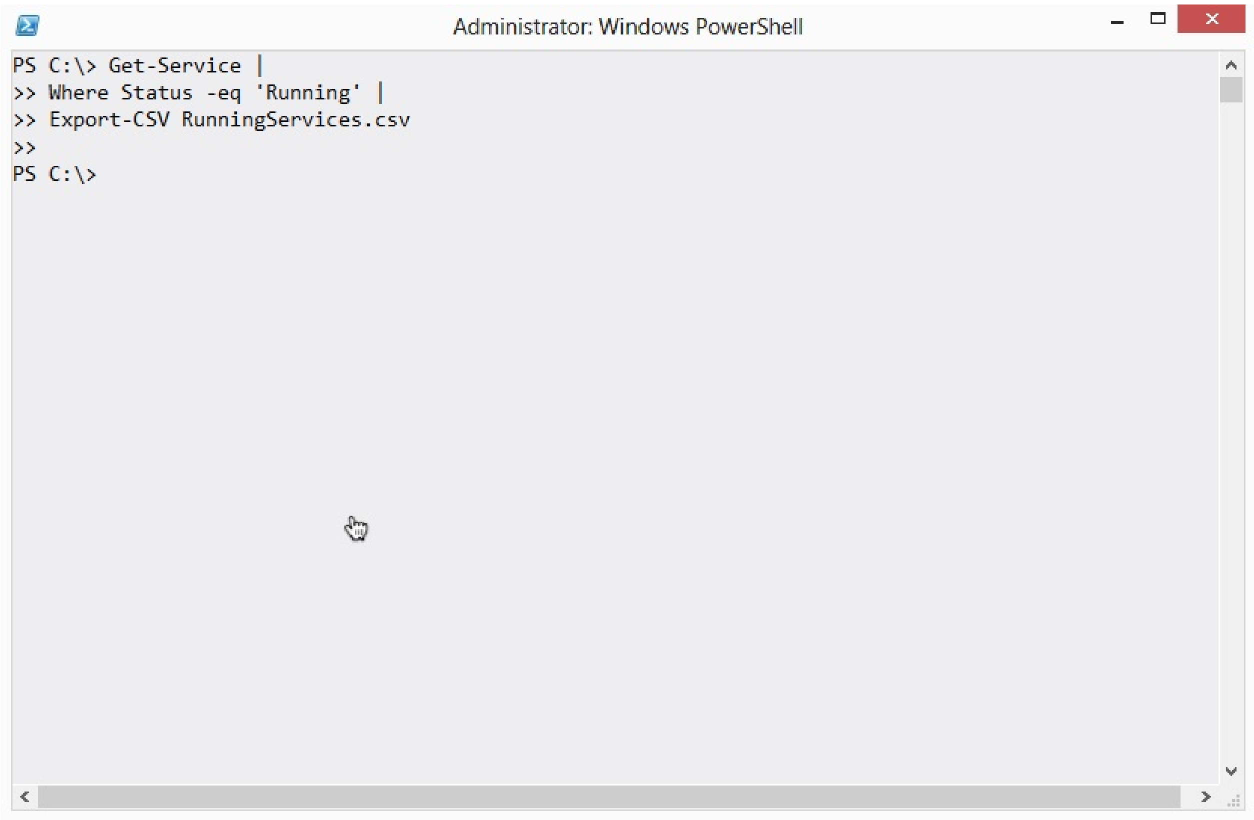
Not Everything Produces Output

I see this one a lot in classes:



If you expected anything on the screen in terms of output, you'd be disappointed. The trick here is to keep track of what each command produces as output, and right there is a possible point of confusion.

In PowerShell's world, output is what would show up on the screen if you ran the command and didn't pipe it to anything else. Yes, Export-CSV does do something - it creates a file on disk - but in PowerShell's world that file isn't output. What Export-CSV does not do is produce any output - that is, something which would show up on the screen. For example:

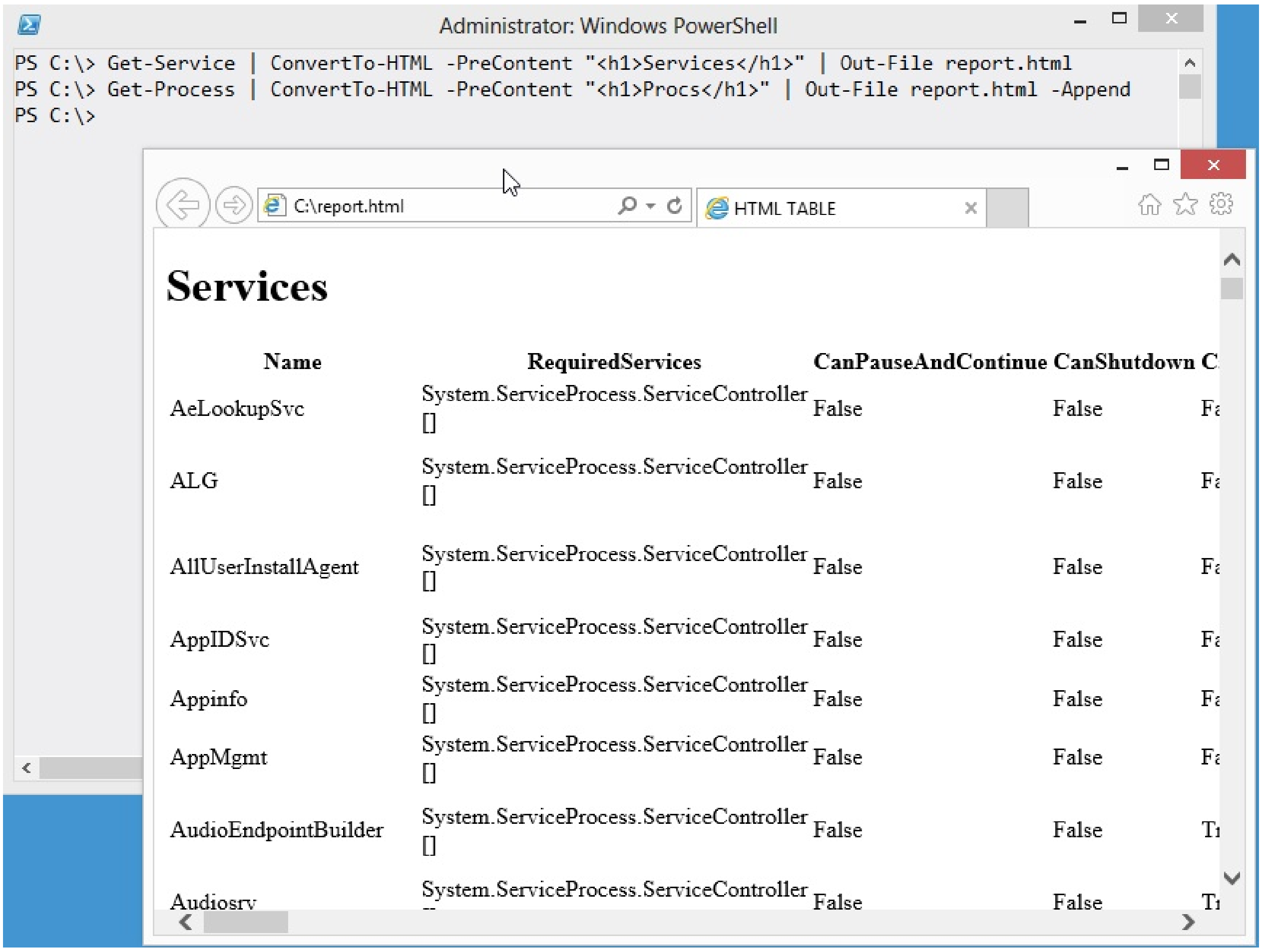


See? Nothing. Since there's nothing on the screen, there's nothing in the pipeline. You can't pipe Export-CSV to another command, because there's nothing to pipe.

Some commands will include a -PassThru parameter. When they have one, and when you use it, they'll do whatever they normally do but also pass their input objects through to the pipeline, so that you can then pipe them on to something else. Export-CSV isn't one of those commands, though - it never produces output, so it will never make sense to pipe it to something else.

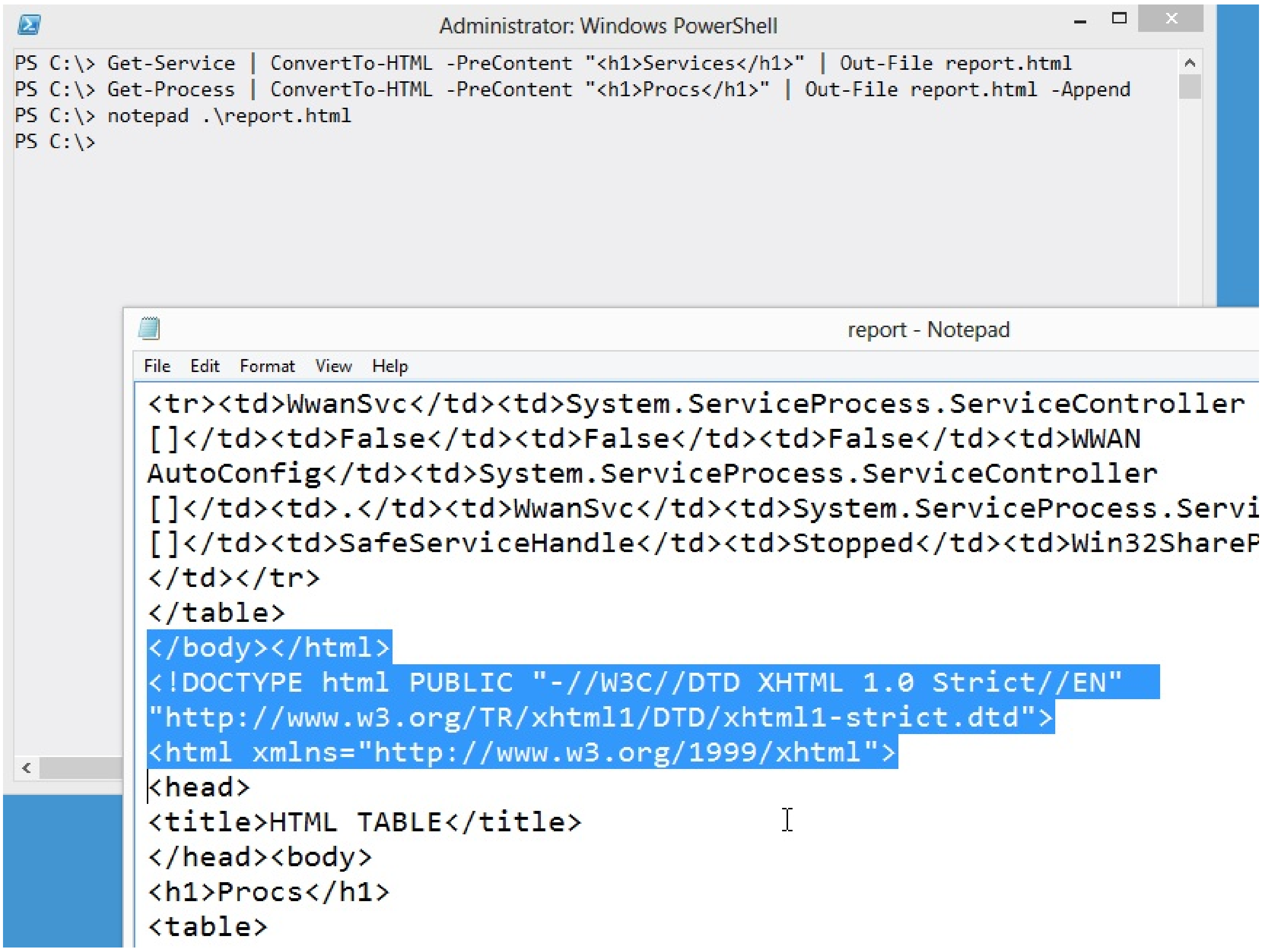
One HTML Page at a Time, Please

This drives me batty:



What's happening is that someone ran two command, piping the output of each to ConvertTo-HTML, and essentially sticking both HTML pages into a single file. What drives me really nuts is that Internet Explorer is okay with that nonsense.

HTML files are allowed to start with one top-level tag, but if you check out that file you'll see that it contains two. Here's the middle bit:

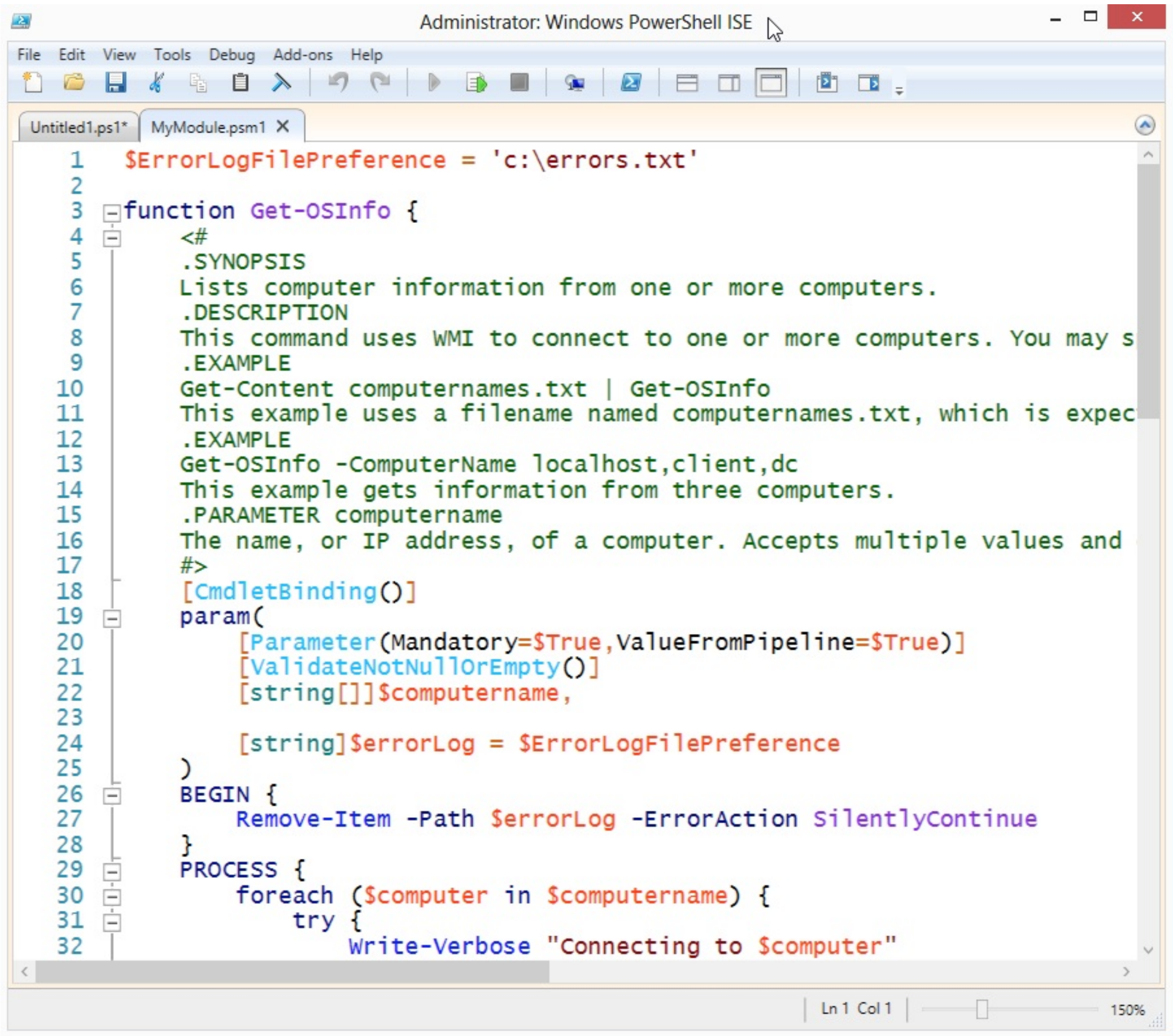


I've highlighted the lines that end one HTML page and start the next one. This is technically a malformed HTML file. It becomes tough to use this with some Web browsers (Firefox 20 is choking it down, but my current Webkit browsers aren't), tough to parse if you ever need to manipulate it programmatically, and... well, it's just a bad thing. It's like incest or something. Gross.

If you need to combine multiple elements into a single HTML file, you use the -Fragment switch of ConvertTo-HTML. That produces just a portion of the HTML, and you can produce several such portions and then combine them into a single, complete page. Ahhh, nice. That whole process is covered in Creating HTML Reports in PowerShell, another free ebook that came with this one

[Bloody] {Awful} (Punctuation)

This isn't so much a "gotcha" as it is just plain confusing. PowerShell's nuts with the punctuation.



(Parentheses) are used to enclose expressions, such as the ForEach() construct's expression, and in certain cases to contain declarative syntax. You see that in the Param() block, and in the [Parameter()] attribute.

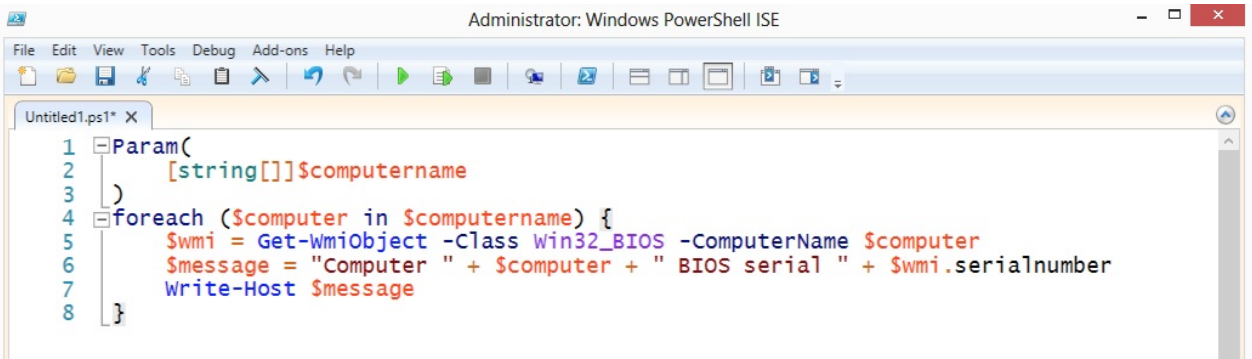
[Square brackets] are used around some attributes, like [CmdletBinding()], and around data types like [string], and to indicate arrays - as in [string[]]. They pop up a few other places, too.

{Curly brackets} nearly always contain executable code, as in the Try{} block, the BEGIN{} block, and the function itself. It's also used to express hash table literals (like @{}).

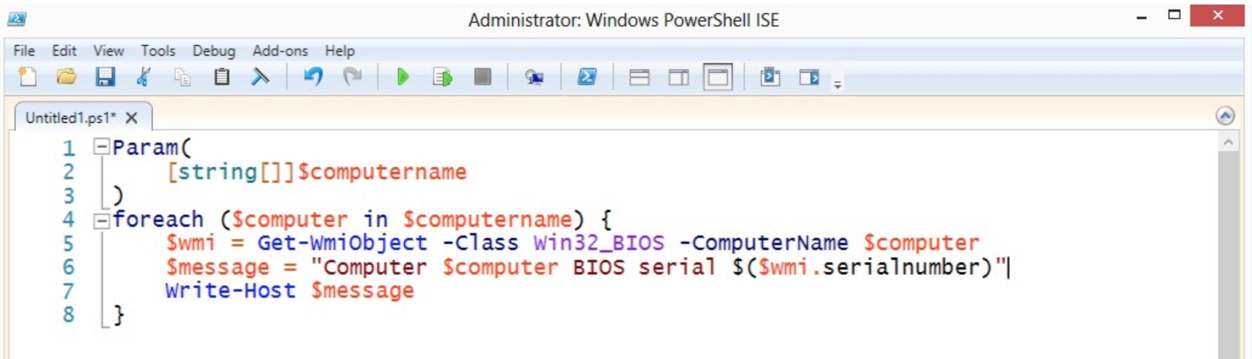
If your keyboard had a few dozen more buttons, PowerShell probably wouldn't have had to have all these overlapping uses of punctuation. But it does. At this point, they're pretty much just part of the shell's "cost of entry," and you'll have to get used to them.

Don’t+Concatenate+Strings

I really dislike string concatenation. It's like forcing someone to cuddle with someone they don't even know. Rude.



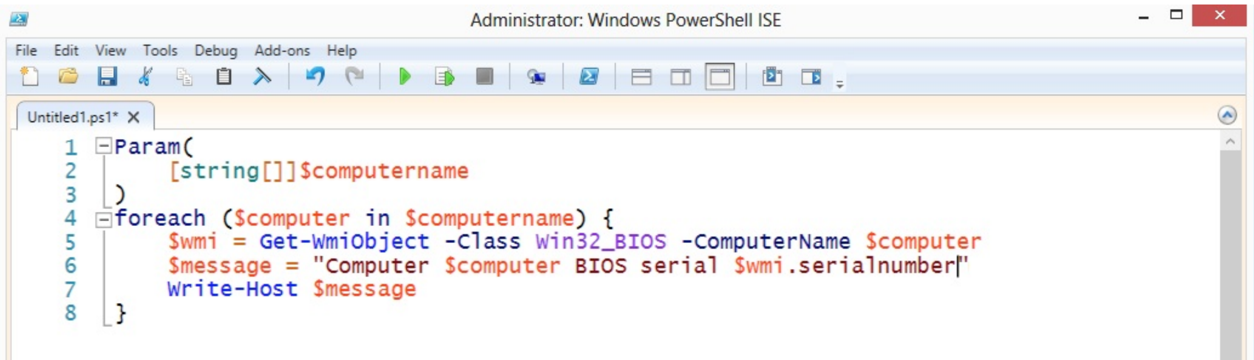
And completely unnecessary, when you use double quotes.



Same end effect. In double quotes, PowerShell will look for the $ character. When it finds it:

1. If the next character is a { then PowerShell will take everything to the matching } as a variable name, and replace the whole thing with that variable's contents. For example, putting ${my variable} inside double quotes will replace that with the contents of ${my variable}.
2. If the next character is a (then PowerShell will take everything to the matching) and execute it as code. So, I executed $wmi.serialnumber to access the serialnumber property of whatever object was in the $wmi variable.
3. Otherwise, PowerShell will take every character that is legal for a variable name, up until the first illegal variable name character, and replace it with that variable. That's how $computer works in my example. The space after r isn't legal for a variable name, so PowerShell knows the variable name stops at r.

There's a sub-gotcha here:



This won't work as expected. In most cases, $wmi will be replaced by an object type name, and .serialnumber will still be in there. That's because . isn't a legal variable name character, so PowerShell stops looking at the variable with the letter i. It replaces $wmi with its contents. You see, in the previous example, I'd put $($wmi.serialnumber), which is a subexpression, and which works. The parentheses make their contents execute as code.

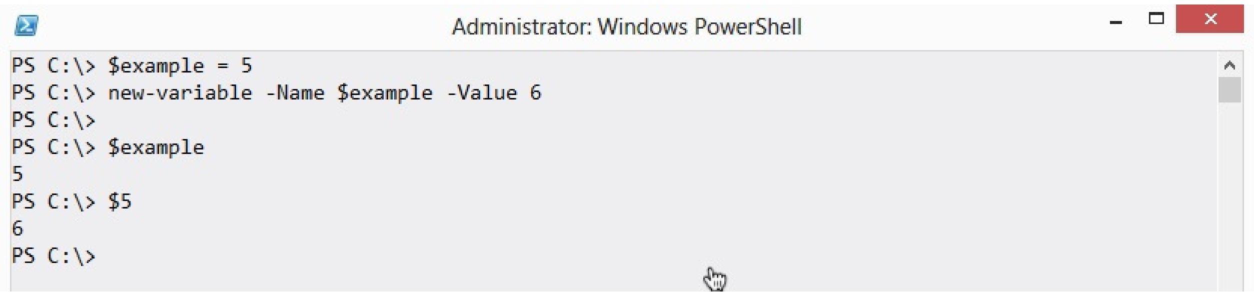
$ isn’t Part of the Variable Name

Big gotcha.




```
PS C:\> $example = 5
PS C:\> new-variable -Name $example -Value 6
PS C:\>
```

Can you predict what happened?

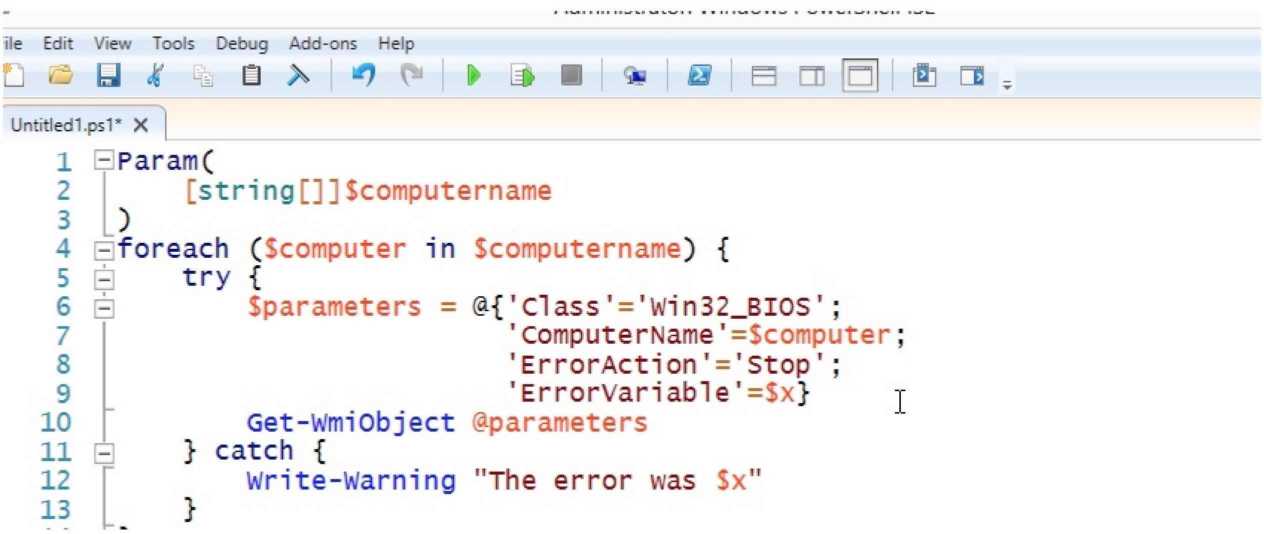



```
PS C:\> $example = 5
PS C:\> new-variable -Name $example -Value 6
PS C:\>
PS C:\> $example
5
PS C:\> $5
6
PS C:\>
```

You see, the $ is not part of the variable's name. If you have a variable named example, that's like having a box with "example" written on the side. Referring to example means you're talking about the box itself. Referring to $example means you're messing with the contents of the box.

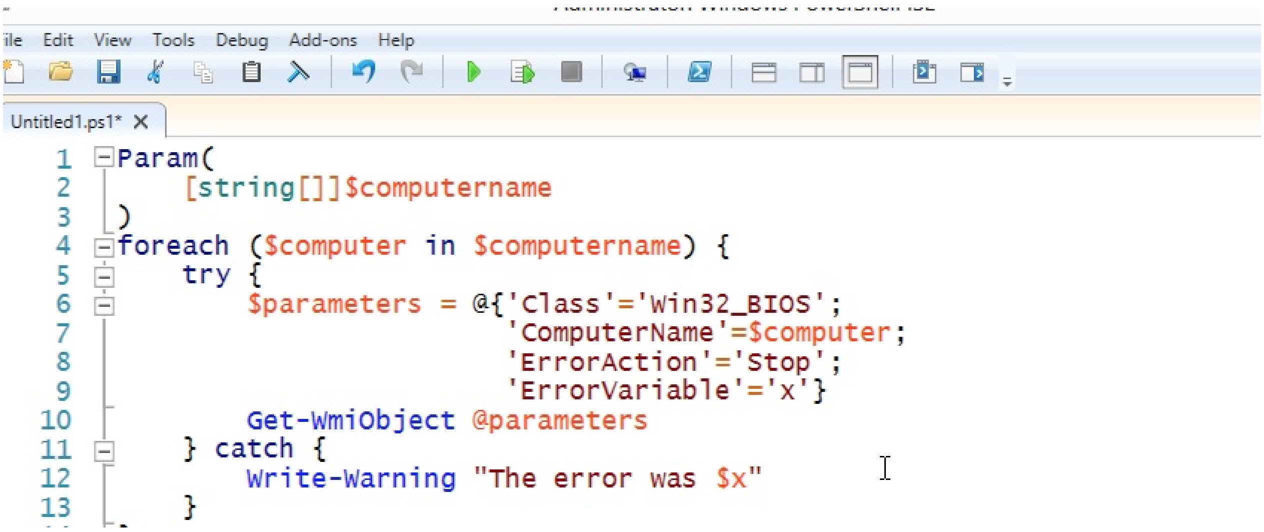
So in my example, I used $example=5 to put 5 into the box. I then created a new variable. The new variable's name was $example - that isn't naming it "example," it's naming it the contents of the "example" box, which is 5. So I create a variable named 5, that contains 6, which you can see by referring to $5.

Tricky, right? Comes up all the time:




```
Param(
    [string[]]$computername
)
foreach ($computer in $computername) {
    try {
        $parameters = @{'Class'='Win32_BIOS';
                        'ComputerName'=$computer;
                        'ErrorAction'='Stop';
                        'ErrorVariable'=$x}
        Get-WmiObject @parameters
    } catch {
        Write-Warning "The error was $x"
    }
```

In that example, I used the -ErrorVariable parameter to specify a variable in which I would store any error that would occur. Problem is, I used $x. I should have used x by itself:



That will store any error in a variable named x, which I can later access by using $x to get its contents - meaning, whatever error was stored in there.

Use the Pipeline, not an Array

A very common mistake made by traditional programmers who come to PowerShell - which is not a programming language:

```
Param(
    [string[]]$computername
)

$output = @()
foreach ($computer in $computername) {
    try {
        $parameters = @{'Class'='Win32_BIOS';
                        'ComputerName'=$computer;
                        'ErrorAction'='Stop';
                        'ErrorVariable'='x'}
        $output += Get-WmiObject @parameters |
                   Select-Object PSComputerName,SerialNumber
    } catch {
        Write-Warning "The error was $x"
    }
}
Write-Output $output
```

This person has created an empty array in $output, and as they run through their computer list and query WMI, they're adding new output objects to the array. Finally, at the end, they output the array to the pipeline.

Poor practice. You see, this forces PowerShell to wait while this entire command completes. Any subsequent commands in the pipeline will sit their twiddling their thumbs. A better approach? Use the pipeline. Its whole purpose is to accumulate output for you - there's no need to accumulate it yourself in an array.

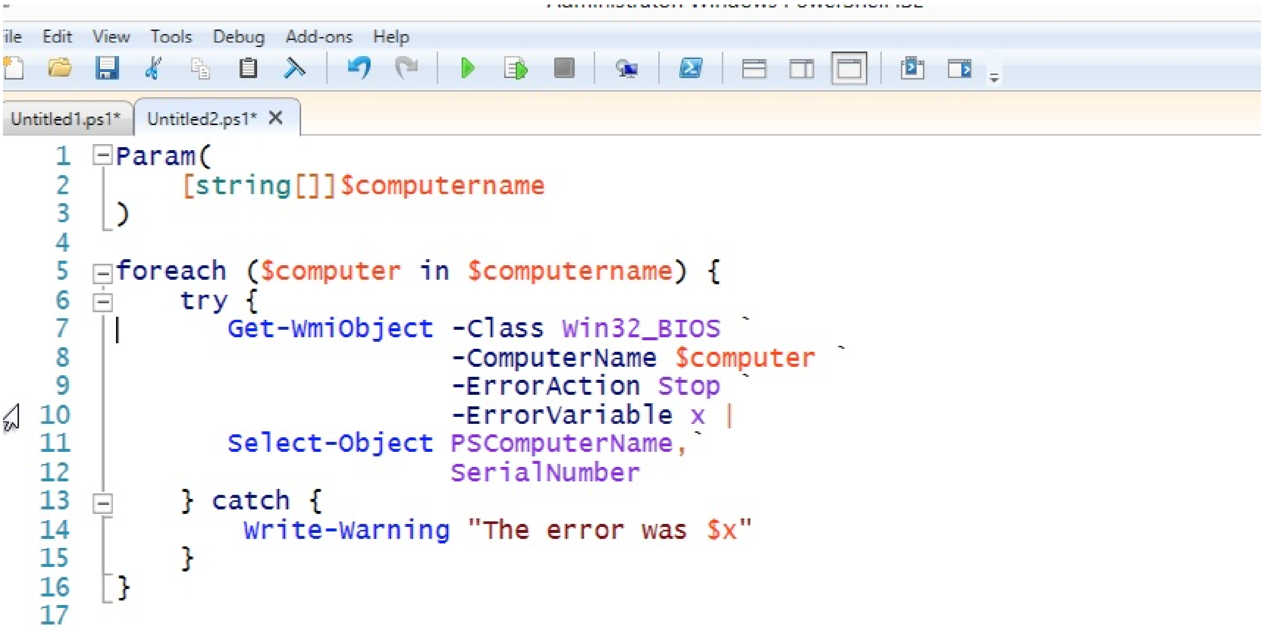
```
Param(
    [string[]]$computername
)

foreach ($computer in $computername) {
    try {
        $parameters = @{'Class'='Win32_BIOS';
                        'ComputerName'=$computer;
                        'ErrorAction'='Stop';
                        'ErrorVariable'='x'}
       Get-WmiObject @parameters |
       Select-Object PSComputerName,SerialNumber
    } catch {
        Write-Warning "The error was $x"
    }
}
```

Now, subsequent commands will receive output as its being created, letting several commands run more or less simultaneously in the pipeline.

Backtick, Grave Accent, Escape

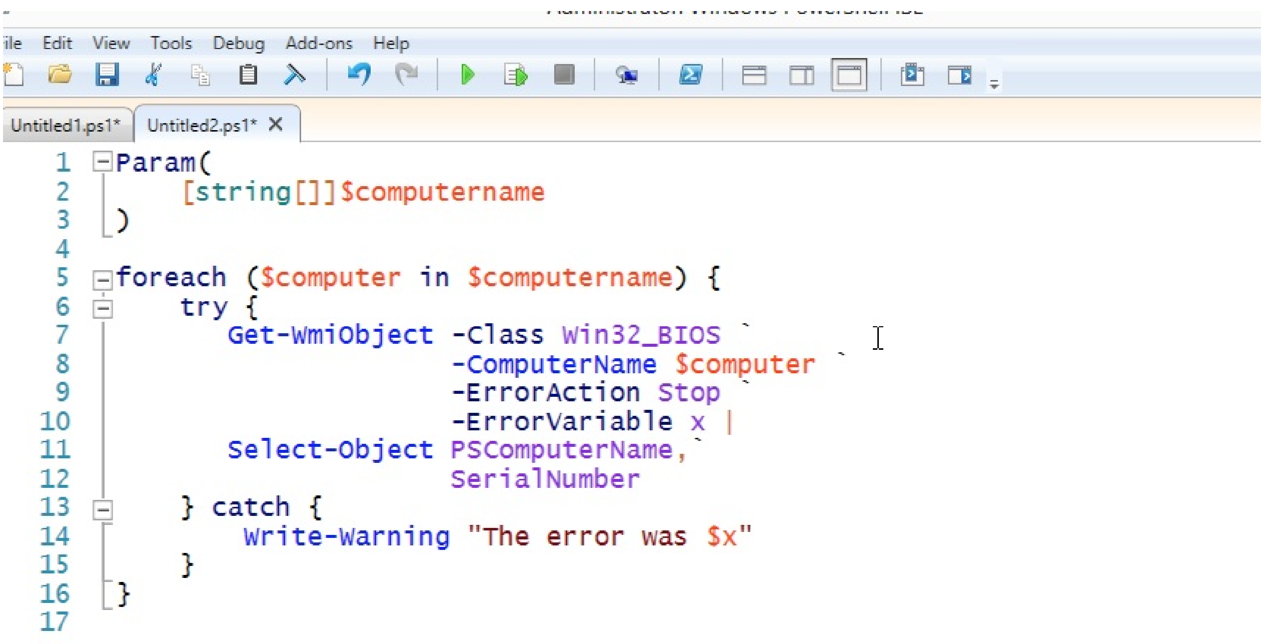
You'll see folks do this a lot:



That isn't a dead pixel on your monitor or a stray piece of toner on the page, it's the grave accent mark or backtick. ` is PowerShell's escape character. In this example, it's "escaping" the invisible carriage return at the end of the line, removing its special purpose as a logical line-end, and simply making it a literal carriage return.

I don't like the backtick used this way.

First, it's hard to see. Second, if you get any extra whitespace after it, it'll no longer escape the carriage return, and your script will break. The ISE even figures this out:

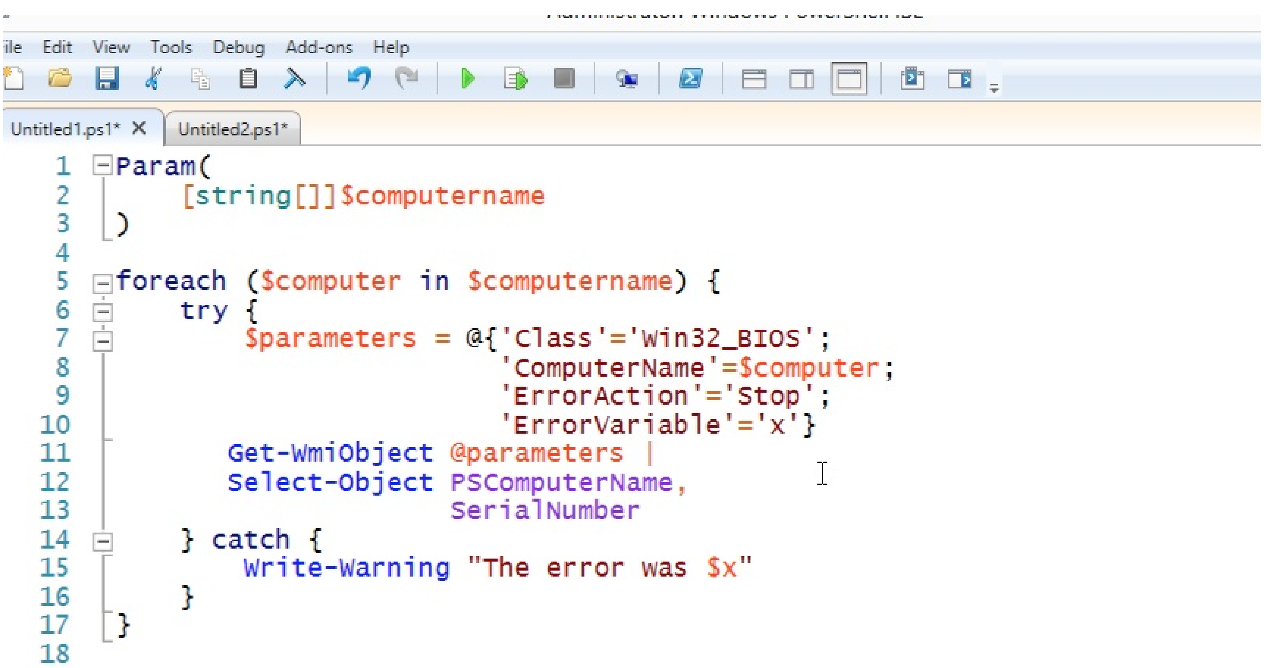


Carefully compare the -ComputerName parameter - in this second example, it's the wrong color for a parameter name, because I added a space after the backtick on the preceding line. IMPOSSIBLE to track these down.

And the backtick is unnecessary as a line continuation character. Let me explain why:

PowerShell already allows you to hit Enter in certain situations. You just have to learn what those situations are, and learn to take advantage of them. I totally understand the desire to have neatly-formatted code - I preach about that all the time, myself - but you don't have to rely on a little three-pixel character to get nicely formatted code.

You just have to be clever.



To begin, I've put my Get-WmiObject commands in a hash table, so I can format them all nice and pretty. Each line ends on a semicolon, and PowerShell lets me line-break after each semicolon. Even if I get an extra space or tab after the semicolon, it'll work fine. I then splat those parameters to the Get-WmiObject command.

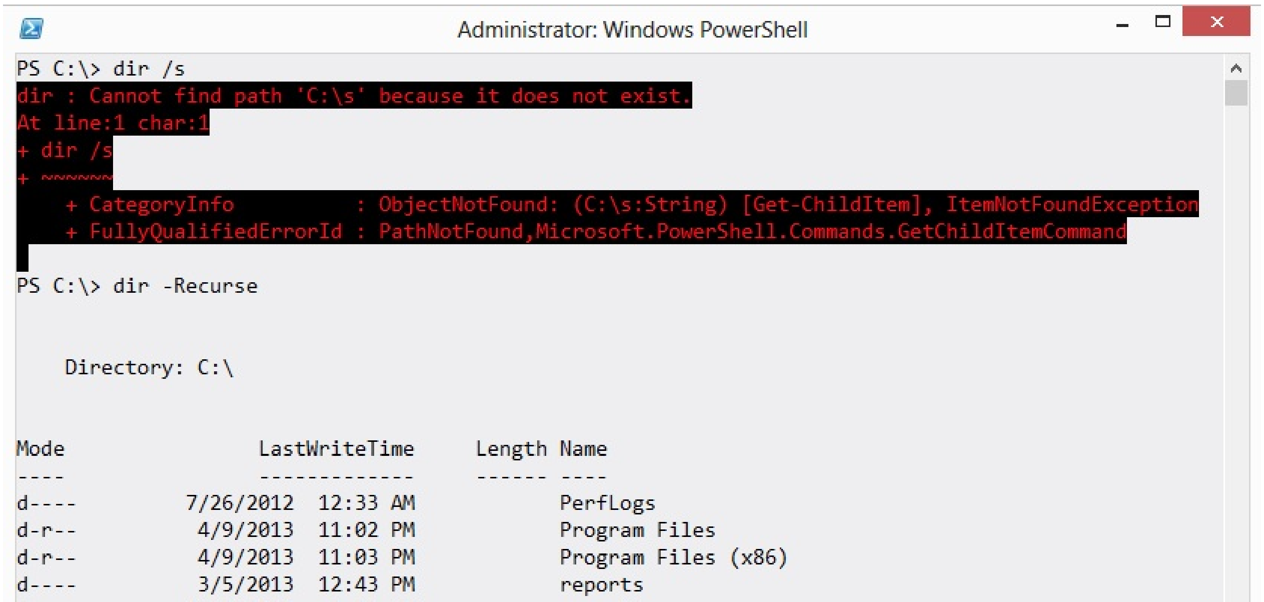
After Get-WmiObject, I have a pipe character - and you can legally line-break after that, too.

You'll notice on Select-Object that breaking after a comma as well.

So I end up with formatting that looks at least as good, if not better, because it doesn't have that little ` floating all over the place.

These aren’t Your Father’s Commands

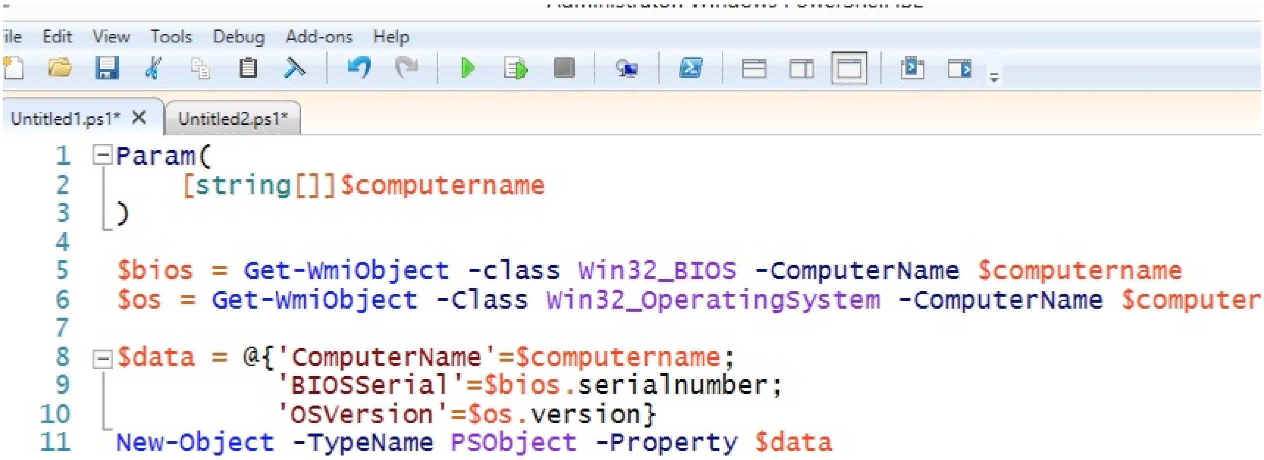
Always keep in mind that while PowerShell has things called Dir and Cd, they aren't the old MS-DOS commands. They're simply aliases, or nicknames, to PowerShell commands. That means they have different syntax.



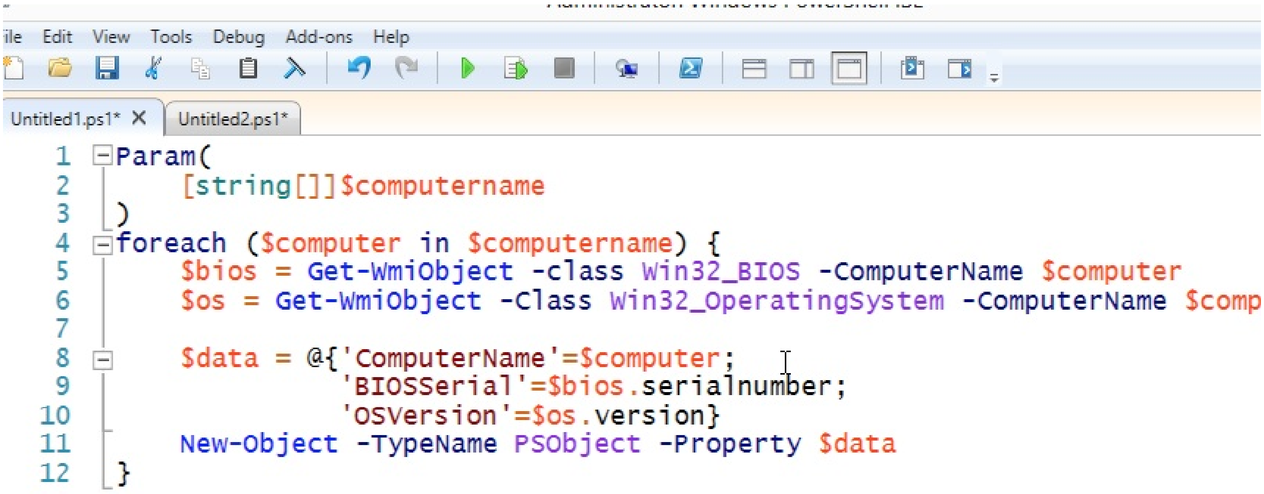
You can run help dir (or ask for help on any other alias) to see the actual command name, and its proper syntax.

A Crowd isn’t an Individual

A very common newcomer mistake:



Here, the person is treating everything like it contains only one value. But $computername might contain multiple computer names (that's what [string[]] means), meaning $bios and $os will contain multiple items too. You'll often have to enumerate those to get this working right:



Folks will run into this even in simple situations. For example:

Administrator: Windows PowerShell

```
PS C:\> $procs = Get-Process
PS C:\> $message = "Process name $procs.name"
PS C:\> $message
Process name System.Diagnostics.Process (conhost) System.Diagnostics.Process (csrss) System.Diagnost
ics.Process (csrss) System.Diagnostics.Process (dwm) System.Diagnostics.Process (explorer) System.Di
agnostics.Process (Idle) System.Diagnostics.Process (lsass) System.Diagnostics.Process (MsMpEng) Sys
tem.Diagnostics.Process (powershell) System.Diagnostics.Process (powershell_ise) System.Diagnostics.
Process (SearchIndexer) System.Diagnostics.Process (services) System.Diagnostics.Process (smss) Syst
em.Diagnostics.Process (spoolsv) System.Diagnostics.Process (svchost) System.Diagnostics.Process (sv
chost) System.Diagnostics.Process (svchost) System.Diagnostics.Process (svchost) System.Diagnostics.
Process (svchost) System.Diagnostics.Process (svchost) System.Diagnostics.Process (svchost) System.D
iagnostics.Process (svchost) System.Diagnostics.Process (svchost) System.Diagnostics.Process (svchos
t) System.Diagnostics.Process (System) System.Diagnostics.Process (TabTip) System.Diagnostics.Proces
s (TabTip32) System.Diagnostics.Process (taskhost) System.Diagnostics.Process (taskhostex) System.Di
agnostics.Process (vmtoolsd) System.Diagnostics.Process (vmtoolsd) System.Diagnostics.Process (winin
it) System.Diagnostics.Process (winlogon).name
PS C:\>
```

PowerShell v2 won't react so nicely; in v3, the variable inside double quotes is $procs, and since that variable contains multiple objects, PowerShell implicitly enumerates them and looks for a Name property. You'll notice ".name" from the original string appended to the end - PowerShell didn't do anything with that.

You'd probably want to enumerate these:

Administrator: Windows PowerShell

```
PS C:\> $procs = Get-Process
PS C:\> $procs | ForEach-Object { "The proc name is $($PSItem.Name)" }
The proc name is conhost
The proc name is csrss
The proc name is csrss
The proc name is dwm
The proc name is explorer
The proc name is Idle
The proc name is lsass
The proc name is MsMpEng
The proc name is powershell
The proc name is powershell_ise
The proc name is SearchIndexer
The proc name is services
The proc name is smss
The proc name is spoolsv
The proc name is svchost
The proc name is svchost
The proc name is svchost
The proc name is svchost
```

Commands' Default Output Can Lie

Well, perhaps not “lie,” but certainly “mislead.”

Try running Get-EventLog -LogName Security on your computer. Notice the column headers in the output?

- The output doesn’t include all of the properties that are available behind the scenes.
- Some of the column headers don’t actually list the correct name for that property.

This can be really frustrating, because if you try to use Select-Object with an incorrect name, it’ll just spit out blanks. The confusion arises because many commands’ output are pre-formatted using a default view. That means you’re not actually seeing the command’s “output,” you’re seeing a “massaged” version of it.

To see the complete output, with the correct property names, run your command and pipe it to `| Format-List *` (or fl *if you prefer). With commands that produce a great deal of output, that can take some time to run and create a messy screen; a shorter version can be obtained by piping your command to `| Select -First 1 | Format-List *`. You’ll see one output object, all of its properties, and the correct property names to use in other commands.

Properties vs. Values

```
$names = Get-ADComputer -filter * |
 Select-Object -Property Name

 Get-CimInstance -Class Win32_BIOS -ComputerName $names
```

Know why that won't work? It's because the result of Get-ADComputer is an object, which has properties. You probably knew that. But the result of Select-Object is also an object that has properties. Specifically, in this case, it's "Selected" ADComputer object, having a single property: Name.

Look at the help for Get-CimInstance. The -ComputerName parameter accepts objects of the type String. It says so, right in the help! But a Selected ADComputer object isn't the same thing as a String. The Name property you selected contains strings, but it isn't a string itself. This is a huge distinction, and one that trips people up all the time.

Think of a property as a box. That box can contain things, but it's a thing in and of itself, also. In this case, the box is called Name, and it contains strings. But you can't shove that whole box into something that was just expecting strings. "Hey, I wanted a string, not a box!"

Think about a fax machine. Do you remember those? They accept pages, and transmit those pages. Now suppose you have an envelope full of pages. You can't just shove the envelope into the fax machine and expect good results. In that analogy, the envelope is a property, and the pages inside it are values. To get the pages into the fax machine, you have to take them out of the envelope first.

What you want to do in this case is get the strings out of the box, and Select-Object offers a way of doing that:

```
$names = Get-ADComputer -filter * |
 Select-Object -ExpandProperty Name

 Get-CimInstance -Class Win32_BIOS -ComputerName $names
```

See the difference? -ExpandProperty gets just the contents of the specified property, rather than returning an object that only has that property. Want a simple way to test this in the shell? Run these commands:

```
Get-Service | Select -Property Name | Get-Member
 Get-Service | Select -ExpandProperty Name | Get-Member
```

Remote Variables

When using PowerShell remoting, you need to remember that you're dealing with two or more computers that don't share information between them. For example, the following command will run fine on your local computer:

```
$f1 = 'D:\Scripts\folder1'
$f2 = 'D:\Scripts\folder2'
Copy-Item -Path $f1 -Recurse -Destination $f2 -Verbose -Force
```

However, if you try to run just the Copy-Item command on a remote computer, it will fail:

```
 $f1 = "D:\Scripts\folder1"
 $f2 = "D:\Scripts\folder2"

 Invoke-Command -ComputerName MemberServer -ScriptBlock {Copy-Item -Path $f1 - Recurse
 -Destination $f2 -Verbose -Force}

 Cannot bind argument to parameter 'Path' because it is null.
 + CategoryInfo : InvalidData: [:] [Copy-Item], ParameterBindingValidationException
 + FullyQualifiedErrorId : ParameterArgumentValidationErrorNullNotAllowed,Microsoft.Po
werShell.Commands.CopyItemCommand
 + PSComputerName : MemberServer
```

The problem here is that $f1 and $f2 are defined on *your* computer, but not on the remote computer. The script block passed by Invoke-Command isn't evaluated on *your* computer, it's simply passed as-is.

There are two possible fixes. The first is to simply include the variable definitions in the script block:

```
 Invoke-Command -ComputerName MemberServer -ScriptBlock {
 $f1 = "D:\Scripts\folder1"
 $f2 = "D:\Scripts\folder2"
 Copy-Item -Path $f1 -Recurse -Destination $f2 -Verbose -Force
 }
```

Another technique, available in PowerShell v3 and later, is to use the $using variable designator. PowerShell pre-scans the script block for these, and will pass along your local variable values to the remote computer(s);

```
$f1 = "D:\Scripts\folder1"
$f2 = "D:\Scripts\folder2"

Invoke-Command -ComputerName MemberServer -ScriptBlock {
Copy-Item -Path $using:f1 -Recurse -Destination $using:f2 -Verbose -Force}
```

The special $using: syntax is what makes this version of the command work.

New-Object PSObject vs. PSCustomObject

There’s often some confusion in regards to the differences between using New-Object PSObject and PSCustomObject, as well as how the two work.

Either approach can be used to take a set of values from a collection of PowerShell objects and collate them into a single output. As well, both avenues will output the data as NoteProperties in the System.Management.Automation.PSCustomObject object types. So what’s the big deal between them?

For starters, the New-Object cmdlet was introduced in PowerShell v1.0 and has gone through a number of changes, while the use of the PSCustomObject class came later in v3.0. For systems using PowerShell v2.0 or earlier, New-Object must be used. The key difference between the 2.0 version and 1.0 version from an administrative point of view is that 2.0 allows the use of hash tables. For example:

New-Object PSObject in v1.0

```
$Path = "c:\scripts"
$Directory = Get-Acl -Path $Path

ForEach ($Dir in $Directory.Access){

    $DirPermissions = New-Object -TypeName PSObject
    $DirPermissions | Add-Member -MemberType NoteProperty -Name Path -Value $Path
    $DirPermissions | Add-Member -MemberType NoteProperty -Name Owner -Value $Director
y.Owner
    $DirPermissions | Add-Member -MemberType NoteProperty -Name Group -Value $Dir.Iden
tityReference
    $DirPermissions | Add-Member -MemberType NoteProperty -Name AccessType -Value $Dir
.AccessControlType
    $DirPermissions | Add-Member -MemberType NoteProperty -Name Rights -Value $Dir.Fil
eSystemRights

    $DirPermissions
}
```

With the New-Object method in PowerShell v1.0, you have to declare the object type you want to create and add members to the collection in individual commands. This changed however in v2.0 with the ability to use hashtables:

New-Object in PS 2.0

```
$Path = "c:\scripts"
$Directory = Get-Acl -Path $Path

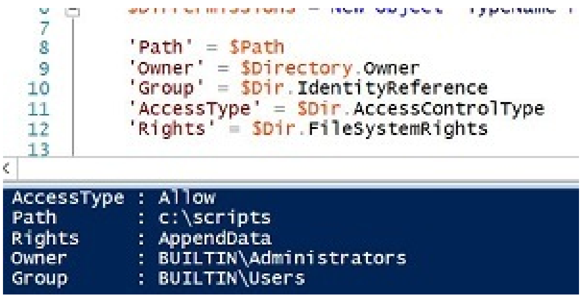
ForEach ($Dir in $Directory.Access){

    $DirPermissions = New-Object -TypeName PSObject -Property @{

    'Path' = $Path
    'Owner' = $Directory.Owner
    'Group' = $Dir.IdentityReference
    'AccessType' = $Dir.AccessControlType
    'Rights' = $Dir.FileSystemRights

    }

    $DirPermissions
}
```

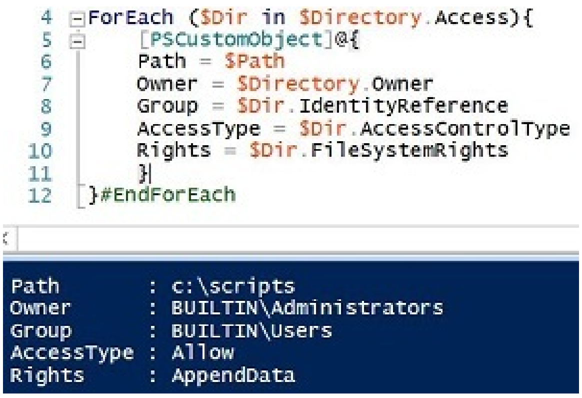
Here's the output: 

This saved a lot of overhead in typing and provided a cleaner looking script. However, both methods have the same problem in that the output is not necessarily in the same order as you have it listed, so if you're looking for a particular format, it may not work. PSCustomObject fixed this when it was introduced in v3.0, along with providing more streamlining in your scripts.

PSCustomObject in PowerShell v3.0

```
$Path = "c:\scripts"
$Directory = Get-Acl -Path $Path

ForEach ($Dir in $Directory.Access){
    [PSCustomObject]@{
    Path = $Path
    Owner = $Directory.Owner
    Group = $Dir.IdentityReference
    AccessType = $Dir.AccessControlType
    Rights = $Dir.FileSystemRights
    }#EndPSCustomObject
}#EndForEach
```



As demonstrated, your output will always match what you have defined in your hashtable. Another advantage of using PSCustomObject is that it has been noted to enumerate the data faster than its New-Object counterpart. The only thing to keep in mind with PSCustomObject is that it will not work with systems running PSv2.0 or earlier.

Running Something as the "Currently Logged-in User"

A common PowerShell request is to be able to remotely kick off some code that runs under the account of the user that’s currently logged on to the remote machine, or the user who most often uses the remote machine.

This is really difficult, and usually impractical.

First, understand that Windows is inherently a multi-user operating system. It doesn’t have a concept for “the currently logged-on user” because there might be many logged-on users. Even though client versions of Windows don’t technically permit multiple interactive logons, the base operating system acts as if it can.

Second, as a multi-user OS, Windows’ job is to maintain a strict firewall around each user’s process space. You don’t want one user jumping into another’s space, because that would be a huge risk to security and stability. So you can’t easily log in as one user and run something that another user can “see.”

For example, a common version of this request is for an admin to remotely make Notepad pop up in front of users, so they can remotely convey some important message. Sadly, Notepad is not a good instant messaging app, and Windows doesn’t make this easy. And, if you think about it, what would malware be able to do if this was possible? It’d be horrible!

With very few, difficult exceptions, you can’t really run something “as another user on a remote machine.” One exception is if you know the remote user’s user name and password. If you do, you can establish a Remoting session to the computer using their credentials, and potentially have applications run in that user’s process space. But you can see how impractical that is in most situations.

Commands that Need a User Profile May Fail When Run Remotely

Many commands act against the currently logged-on user's profile. Those commands can sometimes fail when you run them over a Remoting connection, such as by using Invoke-Command or Enter-PSSession. For example, many installers default to creating per-user icons, and those can fail when run remotely – even when run in a "silent install" mode.

The problem is that, when you connect to a remote computer, you aren't spinning up a complete user environment. You're technically not "logging on" to the machine in the usual sense. You're authenticating, yes, but in much the same way that you'd authenticate to a shared folder. Your remote connection doesn't have a complete user profile, and so anything that's expecting one can get errors and fail (even if they don't show those errors).

There's no easy fix for this, unfortunately.

Writing to SQL Server

Saving data to SQL Server - versus Excel or some other contraption - is easy.

Assume that you have SQL Server Express installed locally. You've created in it a database called MYDB, and in that a table called MYTABLE. The table has ColumnA and ColumnB, which are both strings (VARCHAR) fields. And the database file is in c:\myfiles\mydb.mdf. This is all easy to set up in a GUI if you download SQL Server Express "with tools" edition. And it's free!

```
$cola = "Data to go into ColumnA"
$colb = "Data to go into ColumnB"

$connection_string = "Server=.\SQLExpress;AttachDbFilename=C:\Myfiles\mydb.mdf;Databas
e=mydb;Trusted_Connection=Yes;"
$connection = New-Object System.Data.SqlClient.SqlConnection
$connection.ConnectionString = $connection_string
$connection.Open()
$command = New-Object System.Data.SqlClient.SqlCommand
$command.Connection = $connection

$sql = "INSERT INTO MYTABLE (ColumnA,ColumnB) VALUES('$cola','$colb')"
$command.CommandText = $sql
$command.ExecuteNonQuery()

$connection.close()
```

You can insert lots of values by just looping through the three lines that define the SQL statement and execute it:

```
$cola = @('Value1','Value2','Value3')
$colb = @('Stuff1','Stuff2','Stuff3')

$connection_string = "Server=.\SQLExpress;AttachDbFilename=C:\Myfiles\mydb.mdf;Databas
e=mydb;Trusted_Connection=Yes;"
$connection = New-Object System.Data.SqlClient.SqlConnection
$connection.ConnectionString = $connection_string
$connection.Open()
$command = New-Object System.Data.SqlClient.SqlCommand
$command.Connection = $connection

for ($i=0; $i -lt 3; $i++) {
  $sql = "INSERT INTO MYTABLE (ColumnA,ColumnB) VALUES('$($cola[$i])','$($colb[$i])')"
  $command.CommandText = $sql
  $command.ExecuteNonQuery()
}

$connection.close()
```

It's just as easy to run UPDATE or DELETE queries in exactly the same way. SELECT queries use ExecuteReader() instead of ExecuteNonQuery(), and return a SqlDataReader object that you can use to read column data or advance to the next row.

Getting Folder Sizes

Folks often ask how to use PowerShell to get the size of a folder, such as a user home folder.

Problem is, _folders don't have a size. _Windows literally doesn't track size for folder objects. A folder's "size" is merely the sum of it's files' sizes. Which means you have to add them up.

```
Get-ChildItem -Path <whatever> -File -Recurse |
Measure-Object -Property Length -Sum
```

As one example. Bottom line, you need to get all the files, and add up their Length properties.